The Common Lisp Cookbook

Diving into the programmable programming language

The Common Lisp Cookbook contributors

© 2021 January, 6th, vindarel vindarel@mailz.org. This e-book is free of charge, but you can pay what you want for it.
The Common Lisp Cookbook

Foreword
Getting started
  Install an implementation
    With your package manager
    With Roswell
    With Docker
    On Windows
  Start a REPL
Libraries
  Some terminology
  Install Quicklisp
  Install libraries
  Advanced dependencies management
Working with projects
  Creating a new project
  How to load an existing project
More settings
Read more
Credits
Editor support
  Emacs
    Installing SLIME
    Using Emacs as an IDE
    Setting up Emacs on Windows or Mac
  Vim & Neovim
  Eclipse
  Lem
  Atom
  Sublime Text
  VSCode
  Geany (experimental)
  Notebooks
  REPLs
  Others
Using Emacs as an IDE

Why Use Emacs?

Emacs Lisp vs Common Lisp
- SLIME: Superior Lisp Interaction Mode for Emacs
- SLY: Sylvester the Cat’s Common Lisp IDE

Working with Lisp Code
- Packages for structured editing
- Editing
- Evaluating and Compiling Lisp in SLIME
- Searching Lisp Code

Lisp Documentation in Emacs - Learning About Lisp Symbols
- Argument lists
- Documentation lookup
- Inspect
- Macroexpand
- Consult the CLHS offline

Miscellaneous
- Synchronizing packages
- Calling code
- Exporting symbols
- Project Management

Questions/Answers
- utf-8 encoding
- Default cut/copy/paste keybindings

Appendix
- All Slime REPL shortcuts
- All other Slime shortcuts

See also

Functions
- Named functions: defun

Arguments
- Base case: required arguments
- Optional arguments: &optional
- Named parameters: &key
- Default values to key parameters
- Variable number of arguments: &rest
- Defining key arguments, and allowing more: &allow-other-keys

Return values
- Multiple return values: values, multiple-value-bind and nth-value
Anonymous functions: lambda
Calling functions programmatically: funcall and apply
   Referencing functions by name: single quote ' or sharpsign-quote #’?
Higher order functions: functions that return functions
Closures
setf functions
Currying
   Concept
   With the Alexandria library
Documentation
Strings
   Creating strings
   Accessing Substrings
   Accessing Individual Characters
   Manipulating Parts of a String
   Concatenating Strings
   Processing a String One Character at a Time
   Reversing a String by Word or Character
Dealing with unicode strings
   Sorting unicode strings alphabetically
Controlling Case
   With the format function
Trimming Blanks from the Ends of a String
Converting between Symbols and Strings
Converting between Characters and Strings
Finding an Element of a String
Finding a Substring of a String
Converting a String to a Number
   To an integer: parse-integer
   To any number: read-from-string
   To a float: the parse-float library
Converting a Number to a String
Comparing Strings
String formatting
   Structure of format
   Basic primitive: ~A or ~a (Aesthetics)
   Newlines: ~% and ~&
   Tabs
   Justifying text / add padding on the right
Justifying decimals
Formatting a format string (~v, ~?)
Capturing what is is printed into a stream
Cleaning up strings
    Removing accentuated letters
    Removing punctuation
See also
Numbers
Introduction
    Integer types
    Rational types
    Floating point types
    Complex types
Reading numbers from strings
Converting numbers
    Convert float to rational
    Convert rational to integer
Rounding floating-point and rational numbers
Comparing numbers
Operating on a series of numbers
Working with Roman numerals
Generating random numbers
Bit-wise Operation
Loop, iteration, mapping
Introduction: loop, iterate, for, mapcar, series
Recipes
    Looping forever, return
    Looping a fixed number of times
    Iterate’s for loop
    Looping over a list
    Looping over a vector
    Looping over a hash-table
    Looping over two lists in parallel
    Nested loops
    Computing an intermediate value
    Loop with a counter
    Ascending, descending order, limits
    Steps
    Loop and conditionals
Terminate the loop with a test (until, while)
Loop, print and return a result
Named loops and early exit
Count
Summation
max, min
Destructuring, aka pattern matching against the list or dotted pairs
Custom series scanners
Shorter series expressions
Loop gotchas
Appendix: list of loop keywords
Credit and references
    Loop
    Iterate
    Series
    Others
Multidimensional arrays
Creating
    Random numbers
Accessing elements
    Row major indexing
    Infix syntax
Element-wise operations
    Vectorising expressions
    Calling BLAS
    Reductions
Linear algebra
    Matrix multiplication
    Matrix inverse
    Singular value decomposition
Matlisp
    Creating tensors
    Element access
    Element-wise operations
Dates and Times
    Universal Time
    Internal Time
    Computing the day of the week
    The local-time library
Get today’s date
Formatting time strings
Defining format strings
Parsing time strings

**Pattern Matching**

Common destructuring patterns
- cons
- list, list*
- vector, vector*

Class and structure pattern
- type, satisfies
- assoc, property, alist, plist

Array, simple-array, row-major-array patterns

Logic based patterns
- and, or
- not
- guards

Nesting patterns

See more

**Regular Expressions**

**PPCRE**
- Using PPCRE
- Looking for matching patterns
- Extracting information
- Syntactic sugar

**Input/Output**

Redirecting the Standard Output of your Program

Faithful Output with Character Streams
- CLISP
- CMUCL
- AllegroCL
- LispWorks
- Example

Fast Bulk I/O

**Files and Directories**

Testing whether a file exists
Expanding a file or a directory name with a tilde (~)
Creating directories
Opening a file
Reading files
Writing content to a file
Getting the file extension
Getting file attributes (size, access time, …)
Listing files and directories

Error and exception handling
Ignoring all errors, returning nil
Catching any condition (handler-case)
Catching a specific condition
handler-case VS handler-bind
Defining and making conditions
Signaling (throwing) conditions: error, warn, signal
  Conditions hierarchy
  Custom error messages (:report)
Inspecting the stacktrace
Restarts, interactive choices in the debugger
  Using assert’s optional restart
  Defining restarts (restart-case)
  Changing a variable with restarts
  Calling restarts programmatically (handler-bind, invoke-restart)
  Using other restarts (find-restart)
  Hiding and showing restarts
Handling conditions (handler-bind)
Running some code, condition or not (finally) (unwind-protect)
Conclusion
Resources
See also

Packages
Creating a package
  Accessing symbols from a package
  Importing symbols from another package
  About use-ing packages being a bad practice
List all Symbols in a Package
Package nickname
  Nickname Provided by Packages
  Package locks
See also

Macros
How Macros Work
Quote
Macroexpand
Note: Slime tips
Macros VS functions
Evaluation context
Backquote and comma
Getting Macros Right
Gensym
What Macros are For
See also
Fundamentals of CLOS
Classes and instances
Diving in
Defining classes (defclass)
Creating objects (make-instance)
Slots
find-class, class-name, class-of
Subclasses and inheritance
Multiple inheritance
Redefining and changing a class
Pretty printing
Classes of traditional lisp types
Introspection
See also
Methods
Diving in
Generic functions (defgeneric, defmethod)
Multimethods
Controlling setters (setf-ing methods)
Dispatch mechanism and next methods
Method qualifiers (before, after, around)
Other method combinations
Debugging: tracing method combination
MOP
Metaclasses
Controlling the initialization of instances (initialize-instance)
Type System
Values Have Types, Not Variables
Type Hierarchy
Checking Types
Type Specifier
Defining New Types
Type Checking
Compile-time type checking
  Declaring the type of variables
  Declaring the input and output types of functions
  Declaring &key parameters
  Declaring class slots types
  Alternative type checking syntax: defstar, serapeum
Limitations
See also
TCP/UDP programming with sockets
TCP/IP
UDP/IP
Credit
Interfacing with your OS
Accessing Environment variables
Accessing the command line arguments
  Basics
  Parsing command line arguments
Running external programs
  Synchronously
  Asynchronously
  Input and output from subprocess
Piping
Get Lisp’s current Process ID (PID)
Threads
Introduction
  Why bother?
  What is Concurrency? What is Parallelism?
Bordeaux threads
  Installing Bordeaux Threads
Checking for thread support in Common Lisp
  Basics — list current thread, list all threads, get thread name
Create a thread: print a message onto the top-level
Print a message onto the top-level — read-time eval macro
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
Useful functions
SBCL threads
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 — 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
Useful functions
Wrap-up
Parallel programming with lparallel
Installation
Preamble - get the number of cores
Common Setup
Using channels and queues
Killing tasks
Using promises and futures
Using cognates - parallel equivalents of Common Lisp counterparts
Error handling
Monitoring and controlling threads with Slime
References
Defining Systems
ASDF
Simple examples
Loading a system
Testing a system
Designating a system
How to write a trivial system definition
How to write a trivial testing definition
Create a project skeleton
Debugging
Print debugging
Logging
Using the powerful REPL
Inspect and describe
The interactive debugger
Trace
  Tracing method invocation
Step
Break
  Breakpoints in Slime
Advise and watch
Unit tests
Remote debugging
References
Performance Tuning and Tips
Finding Bottlenecks
  Acquiring Execution Time
  Know your Lisp’s statistical profiler
  Use flamegraphs and other tracing profilers
  Checking Assembly Code
Using Declare Expression
  Speed and Safety
  Type Hints
  More on Type Declaration with declaim
  Declaring function types
Code Inline
Optimizing Generic Functions
  Using Static Dispatch
Scripting, Command line arguments, Executables
Building a self-contained executable
  With SBCL - Images and Executables
  With ASDF
  With Roswell or Buildapp
For web apps
  Size and startup times of executables per implementation
  Building a smaller binary with SBCL’s core compression
Parsing command line arguments
  Declaring arguments
  Parsing
Continuous delivery of executables
Credit
Testing
Testing with Prove
   Install and load
   Write a test file
   Run a test file
   Run one test
   More about Prove
Interactively fixing unit tests
Code coverage
   Generating an html test coverage output
Continuous Integration
   Travis CI and Coveralls
   Gitlab CI
Database Access and Persistence
The Mito ORM and SxQL
   Overview
   Connecting to a DB
   Models
   Migrations
   Queries
   Triggers
   Inflation/Deflation
   Eager loading
   Schema versioning
   Introspection
Testing
See also
GUI toolkits
Introduction
   Tk (Ltk)
   Qt4 (Qtools)
   Gtk+3 (cl-cffi-gtk)
   IUP (lispnik/IUP)
   Nuklear (Bodge-Nuklear)
Getting started
   Tk
   Qt4
   Gtk3
   IUP
Nuklear

Conclusion

Web development

Overview

Installation

Simple webserver

Serve local files

Access your server from the internet

Hunchentoot

Routing

Simple routes

Accessing GET and POST parameters

Error handling

Hunchentoot

Clack

Weblocks - solving the JavaScript problem©

Templates

Djula - HTML markup

Spinneret - lispy templates

Serve static assets

Hunchentoot

Connecting to a database

Checking a user is logged-in

Encrypting passwords

Running and building

Running the application from source

Building a self-contained executable

Continuous delivery with Travis CI or Gitlab CI

Multi-platform delivery with Electron

Deployment

Deploying manually

Daemonizing, restarting in case of crashes, handling logs with Systemd

With Docker

With Guix

Deploying on Heroku and other services

Monitoring

Connecting to a remote Lisp image

Hot reload

See also
Credits

Web Scraping
  HTTP Requests
  Parsing and extracting content with CSS selectors
  Async requests

WebSockets
  The websocket-driver Concept
  Defining Handlers for Chat Server Logic
  Defining A Server
  A Quick HTML Chat Client
  Check it out!
  All The Code

APPENDIX: Contributors
Foreword

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

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 Emacs25, 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

In doubt, just get 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 probably for your distro:

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

There is also:

- 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 Roswell**

R**oswell** 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 **daewok/lisp-devel-docker** image comes with recent versions of SBCL, CCL, ECL and ABCL, plus Quicklisp installed in the home (/home/lisp) so than we can **ql:quickload** libraries straight away.

It works on GNU/Linux, Mac and Windows.

The following command will download the required image (around 400MB), 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:/usr/local/share/common-l
But we still want to develop from our Emacs and SLIME, so we need to connect SLIME to the Lisp inside Docker. See [slime-docker](https://example.com) for a library that helps on setting that up.

**On Windows**

All implementations above can be installed on Windows.

[Portacle](https://example.com) is multiplatform and works on Windows.

You can also try:

- **pEmacs**, 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)
```
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 one’s editor and to use SBCL or CCL for production.

Libraries

Common Lisp has hundreds 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:

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

and enter a Lisp REPL and load this file:

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

or

2- install the Debian package:

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

and load it, from a REPL:

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

Then, in both cases, still from the REPL:

```lisp
(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 "package-name")

and voilà. See Quicklisp’s documentation for more commands.

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 (for regular expressions), one should first install the cl-ppcre package.
Then, in SBCL and ECL, it can be used with:

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

See more: https://wiki.debian.org/CommonLisp

**Advanced dependencies management**

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

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

For a complete list, see

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

and

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

A library installed here is automatically available for every project.

**Providing our own version of a library. Cloning projects.**

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

```lisp
(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.

**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.

Quicklisp also provides [Quicklisp bundles](https://www.quicklisp.org/bundles.html). They are self-contained sets of systems that are exported from Quicklisp and loadable without involving Quicklisp.

At last, there’s [Quicklisp controller](https://quicklispcontroller.readthedocs.io/) 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](https://github.com/clproject-org/cl-project) that also sets up a tests skeleton.

In short:

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

it will create a directory structure like this:
Where `my-project.asd` resembles this:

```
(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.

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:

```
(pushnew "~/path-to-project/root/" asdf:*central-registry* :test #'equal)
```

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

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)
```

Beware that this might break some packages like Mito. Avoid doing this in production.

**Read more**

- 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 Emacs25, 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 \texttt{M-x slime} and/or \texttt{M-x slime-connect}.

See also:

\begin{itemize}
  \item \url{https://wikemacs.org/wiki/SLIME} - configuration examples and extensions.
\end{itemize}

\textbf{Using Emacs as an IDE}

See \textit{“Using Emacs as an IDE”}.

\textbf{Setting up Emacs on Windows or Mac}

See \textit{“Setting up Emacs on Windows or Mac”}.

\textbf{Vim & Neovim}

\texttt{Slimv} is a full-blown environment for Common Lisp inside of Vim.

\texttt{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.

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

**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 Electron frontend, and other programming modes: Python, Go, Rust, JS, Nim, Scheme, HTML, CSS, directory mode, a vim layer, and more.
Atom

See SLIMA. This package allows you to interactively develop Common Lisp code, turning Atom into a pretty good Lisp IDE.
**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.

---

**VSCode**

*VSCode* with *commonlisp-vscode extension* supports running a REPL, evaluate code, auto indent, code completion, go to definition, documentation on hover, etc. It’s based on *cl-lsp* language server and it’s possible to write LSP client that works in other editors.
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.
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

For reviews of plugins for more editors, including free versions of proprietary ones (LispWorks, Allegro), see Articulate Common Lisp.
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 ,q1 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.

**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:
Put the cursor on the open parens of "(defun ..." and press "C-M-q" to indent the code:

```lisp
(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 ((())), M-x check-parens to spot malformed sexps, C-u <n> M-( to enclose sexps with parens:

For example (point is before the parenthesis):

```lisp
|(- 2 2)
;; Press C-u 1 M-( to enclose it with parens:
||(- 2 2))
```

**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),
- evaluate a region with `C-c C-r`,
- evaluate a defun with `C-M-x`.

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:

```lisp
(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:
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 \texttt{M-. (slime-edit-definition)} to go to its definition. Press \texttt{M-}, to come back.

**Go to symbol, list symbols in current source**

Use \texttt{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 i	exttt{menu 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 a nice cross referencing facility, for example, you can see what calls a particular function or expands a macro. It presents a list of places which reference a particular entity, from there you can recompile the thing which references by pressing \texttt{C-c C-c} on that line. \texttt{C-c C-k} will recompile all the references. This is useful when modifying macros, inline functions, or constants.

The following bindings are also shown in Slime’s menu:

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

And when the slime-asdf contrib is enabled, \texttt{C-c C-w d (slime-who-depends-on)} lists dependent ASDF systems

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

(thanks to Slime tips)

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

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 CLHS offline
(ql:quickload "clhs")

Then add this to your Emacs configuration:

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

**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)

```lisp
(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" n)))
```
(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](#)

**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:

```
(setq 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)
,
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 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
See also
See also

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

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 arg

(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.

```
(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:

```
(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:

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

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

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

**Mixing optional and key parameters**

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

```
(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
;    ; caught STYLE-WARNING:
;    ;   &OPTIONAL and &KEY found in the same lambda list: (&OPTIONAL (NA
;    ;
;    ; 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.

**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)
      (+ (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
**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.

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

This function returns a.

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

We use `values` to return multiple values:

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

```lisp
(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 destruct 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:

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

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

```
(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
;;
;; But:

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

#' is actually the shorthand for (function ...):

```
(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.

**Higher order functions: functions that return functions**
Writing functions that return functions is simple enough:

\[
\begin{align*}
\text{(defun adder (n) } \quad \\
\text{ (lambda (x) (+ x n))))} \\
\text{ ;; ADDER}
\end{align*}
\]

Here we have defined the function adder which returns an object of type function.

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

\[
\begin{align*}
\text{(adder 5)} \\
\text{ ;; #<CLOSURE (LAMBDA (X) :IN ADDER) \{100994ACDB\>}
\end{align*}
\]

\[
\begin{align*}
\text{(funcall (adder 5) 3)} \\
\text{ ;; 8}
\end{align*}
\]

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

\[
\begin{align*}
\text{((adder 3) 5)} \\
\text{In: (ADDER 3) 5} \\
\text{ ((ADDER 3) 5)} \\
\text{Error: Illegal function call.}
\end{align*}
\]

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.

\[
\begin{align*}
\text{;; The symbol foo is bound to nothing:}
\text{ CL-USER> (boundp 'foo)} \\
\text{NIL} \\
\text{ CL-USER> (fboundp 'foo)} \\
\text{NIL} \\
\text{ ;; We create a variable:}
\text{ CL-USER> (defparameter foo 42)} \\
\text{ FOO} \\
\text{ * foo} \\
\text{ 42} \\
\text{ ;; Now foo is "bound":}
\text{ CL-USER> (boundp 'foo)}
\end{align*}
\]
;; 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`):

```lisp
;;; 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:

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

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

This mechanism is particularly used for CLOS methods.

A silly example:

```
(defun hello (name)
```

See more on [Practical Common Lisp](https://www实战commonlisp.com).

**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:
(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 argument.

(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#functions
- ordinary lambda lists: http://www.lispworks.com/documentation/HyperSpec/Body/03_da.htm
- multiple-value-bind: http://clhs.lisp.se/Body/m_multip.htm
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
- plus a Slime tip: type C-c C-d ~ plus a letter of a format directive to open up its documentation. 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)
#
* (char *my-string* 7)
#
* (char *my-string* 6)
#
* (setf (char *my-string* 6) #\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 1488)
  #\HEBREW LETTER ALEF

  * (char-code #\HEBREW LETTER ALEF)
1488
Check out the UTF-8 Wikipedia article for the range of supported characters and their encodings.

**Manipulating Parts of 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 '#\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"

* `(substitute '#\u '#\o "Groucho Marx")`
  "Gruuchu Marx"

* `(substitute-if '#_ #'upper-case-p "Groucho Marx")`
  "_rousco _arx"

* `(defparameter *my-string* (string "Zeppo Marx"))`
  *MY-STRING*

* `(replace *my-string* "Harpo" :end 5)`
  "Harpo Marx"

* `*my-string*`
  "Harpo Marx"

Another function that can be frequently used (but not part of the ANSI standard) is replace-all. This function provides an easy functionality for search/replace operations on a string, by returning a new string in which all the occurrences of the ‘part’ in string is replaced with ‘replacement’.

* `(replace-all "Groucho Marx Groucho" "Groucho" "ReplacementForGroucho"
"ReplacementForGroucho Marx ReplacementForGroucho"

One of the implementations of replace-all is as follows:
(defun replace-all (string part replacement &key (test #'char=))
 "Returns a new string in which all the occurrences of the part
 is replaced with replacement."
 (with-output-to-string (out)
   (loop with part-length = (length part)
         for old-pos = 0 then (+ pos part-length)
         for pos = (search part string
                      :start2 old-pos
                      :test test)
     do (write-string string out
         :start old-pos
         :end (or pos (length string)))
     when pos do (write-string replacement out
                 while pos)))

However, bear in mind that the above code is not optimized for long strings; if
you intend to perform such an operation on very long strings, files, etc. please
consider using cl-ppcre regular expressions and string processing library which
is heavily optimized.

**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 '(
    
    (princ char stream))
  )
"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 #\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."

```
(\texttt{loop \ for \ i = 0 \ then \ (1+ j)}
  \texttt{as \ j = (position \ #\Space \ string \ :start \ i)}
  \texttt{collect \ (subseq \ string \ i \ j)}
  \texttt{while \ j))}
```

\texttt{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 ")
*(\texttt{defun join-string-list \ (\texttt{string-list})})
"Concatenates a list of strings and puts spaces between the elements."
(\texttt{format \ nil \ "\{-A-^-\}" \ string-list})
```
\texttt{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
  (\texttt{reverse}
    (split-by-one-space
      "Reverse this sentence by word")))
"word by sentence this Reverse"
```

\textbf{Dealing with unicode strings}

We’ll use here \texttt{SBCL’s string operations}. More generally, see \texttt{SBCL’s unicode support}.

\textbf{Sorting unicode strings alphabetically}

Sorting unicode strings with \texttt{string-lessp} as the comparison function isn’t satisfying:
(sort '("Aaa" "Ééé" "Zzz") #'string-lessp)
;; ("Aaa" "Zzz" "Ééé")

With SBCL, use sb-unicode:unicode<:

(sort '("Aaa" "Ééé" "Zzz") #'sb-unicode:unicode<)
;; ("Aaa" "Ééé" "Zzz")

## 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-string*))
  *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-DRIGHT-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"
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.

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 cannot 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)
"xxxxxx"
* (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.

* (find '#' "The Hyperspec contains approximately 110,000 hyperlinks"
* (find '#' "The Hyperspec contains approximately 110,000 hyperlinks"
* (find '#' "The Hyperspec contains approximately 110,000 hyperlinks" NIL
* (find-if #'digit-char-p "The Hyperspec contains approximately 110,000 hyperlinks"
* (find-if #'digit-char-p "The Hyperspec contains approximately 110,000 hyperlinks"
* (position '#' "The Hyperspec contains approximately 110,000 hyperlinks"
  17
* (position '#' "The Hyperspec contains approximately 110,000 hyperlinks"
  0
* (position-if #'digit-char-p "The Hyperspec contains approximately 110,000 hyperlinks"
  37
* (position-if #'digit-char-p "The Hyperspec contains approximately 110,000 hyperlinks"
  43

Or use COUNT and friends to count characters in a string.

* (count '#' "The Hyperspec contains approximately 110,000 hyperlink"
  2
* (count '#' "The Hyperspec contains approximately 110,000 hyperlink"
  3
* (count-if #'digit-char-p "The Hyperspec contains approximately 110,000 hyperlink"
  6
* (count-if #'digit-char-p "The Hyperspec contains approximately 110,000 hyperlink"
  5

Finding a Substring of a String

The function SEARCH can find substrings of a string.
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.

**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.

```lisp
* (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.

```lisp
(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:

```lisp
(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 it)))
movies)
```

which prints:

```
1 fooo baaar 5
10 last 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 comma-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.

**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) (third it)
      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) (third it)
      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@" "yo")

(mapcar (lambda (it)
            (format nil "~2@a ~25@a ~2a~%" (first it) (second it) (third movies)))
        [(1 "Matrix 5") (10 "Matrix Trilogy swe sub 3.3")])

Justifying decimals

In \(\sim 2F\), 2 is the number of decimals and F the floats directive: (format t "\sim 2F\ 20.1") \(\Rightarrow\) “20.10”.

With \(\sim2,2f\):

(mapcar (lambda (it)
            (format t "\sim2@a \sim25a \sim2,2f~%" (first it) (second it) (third movies)))
        [(1 "Matrix") (10 "Matrix Trilogy swe sub 3.3")])

And we’re happy with this result.

Formatting a format string (\(\sim v\), \(\sim ?\))

Sometimes you want to justify a string, but the length is a variable itself. You can’t hardcode its value as in (format nil "\sim30a" "foo"). Enters the \(v\) directive. We can use it in place of the comma-separated prefix parameters:

(let ((padding 30))
     (format nil "\sim va" padding "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"))
```

**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.~" o)
  ;; Output is: 'hello you'. It is indeed a (SIMPLE-ARRAY CHARACTER (9
  ;; NIL
```

**Cleaning up strings**

The following examples use the cl-slug library which, internally, iterates over the characters of the string and uses ppcnet: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 . "Suom"  
(:UK . "українська (Ukrainian)") (:RU . "Русский (Russian)") (:RO .  
(:RM . "Румынч (Romansh)") (:PT . "Português (Portuguese)") (:PL  
(:NO . "Norsk (Norwegian)") (:LT . "Lietuvių (Lithuanian)") (:LV .  
(:LA . "Lingua Latina (Latin)") (:IT . "Italiano (Italian)") (:EL .  
(:DE . "Deutsch (German)") (:DA . "Dansk (Danish)") (:CS . "Čeština  
(: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 pcre unicode regexp (pcre:regex-replace-all "[^\p{L}\p{N}]+" where \p{L} is the “letter” category and \p{N} any kind of numeric character).

See also

- [Pretty printing table data](#), in ASCII art, a tutorial as a Jupyter notebook.
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: 

```
.lisp * (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-ZERO

* (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)
(:TRAPS (:OVERFLOW :INVALID :DIVIDE-BY-ZERO) :ROUNDING-MODE :NEAREST
```
Arbitrary precision

For arbitrary high precision calculations there is the `computable-reals` library on QuickLisp:

```lisp
* (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`. 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: (+ 1
* (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

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)`
Comparing numbers


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:

* (= (+ 1e0 5e-8) 1e0)
  T
* (= (+ 1e0 6e-8) 1e0)
  NIL

Note that this does not mean that a single-float is always precise to within 6e-8:

* (= (+ 10e0 4e-7) 10e0)
  T
* (= (+ 10e0 5e-7) 10e0)
  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](https://gist.github.com/tormaroe/76795) 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.

```lisp
* (random 10) 7
```

```lisp
* (type-of (random 10))
(INTEGER 0 4611686018427387903)
```

```lisp
* (type-of (random 10.0))
SINGLE-FLOAT
```

```lisp
* (type-of (random 10d0))
DOUBLE-FLOAT
```

In SBCL a [Mersenne Twister](https://en.wikipedia.org/wiki/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*`:

```lisp
* (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.

```plaintext
<table>
<thead>
<tr>
<th>Common Lisp</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>(logand a b c)</td>
<td>a &amp; b &amp; c</td>
</tr>
<tr>
<td>(logior a b c)</td>
<td>a</td>
</tr>
<tr>
<td>(lognot a)</td>
<td>~a</td>
</tr>
<tr>
<td>(logxor a b c)</td>
<td>a ^ b ^ c</td>
</tr>
<tr>
<td>(ash a b)</td>
<td>a &lt;&lt; b</td>
</tr>
<tr>
<td>(ash a -b)</td>
<td>a &gt;&gt; b</td>
</tr>
</tbody>
</table>
```

Negative numbers are treated as two’s-complements. If you have forgotten this, please refer to the [Wiki page](#).

For example:

```lisp
* (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)
```
* (logxor 1 3 7 15)
10

* (lognot -1)
0

* (lognot -3)
2

* (ash 3 2)
12

* (ash -5 -2)
-2

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 var
;; FOR FIRST THEN Set var on first, and then on subsequen
;; ... 

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))
;; (1 2 3)
(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

`doimes`

```
(dotimes (n 10)
  (print n))
```

Here `doimes` returns `nil`. The return value is evaluated at the end of the loop.

You can use `return` inside of it:

```
(dotimes (i 10)
  (if (-> i 3)
    (return)
    (print i)))
```

`loop... repeat`

```
(loop repeat 10
  do (format t "Hello!~\n"))
```

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))
```

`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))
```

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:

```lisp
(loop for x in '(a b c)
  do (print x))
;; A
;; B
;; C
;; NIL
```

```lisp
(loop for x in '(a b c)
  collect x)
;; (A B C)
```

With on, we loop over the cdr of the list:

```lisp
(loop for i on '(1 2 3) do (print i))
;; (1 2 3)
;; (2 3)
;; (3)
```

**mapcar**

```lisp
(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**

```lisp
(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:

(setf h (make-hash-table))
(setf (gethash 'a h) 1)
(setf (gethash 'b h) 2)

**loop**

Looping over keys:

(loop for k being the hash-key of h do (print k))

;;; b
;;; a

same with hash-value.
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

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:

```
(maphash (lambda (key val)
            (format t "key: ~a val:~a~&" key val))
        h)
```

See also `with-hash-table-iterator`.

Series

```
(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))
```

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
A more efficient way, when the lists are known to be of equal length:

\[
\text{(collect} \\
\text{(mapping } (((x \ y) (scan-multiple 'list} \\
\text{ '(a b c) } \\
\text{ '(1 2 3))))} \\
\text{(list x y)))}
\]

Return a flat list:

\[
\text{(collect-append } ; \text{ or collect-nconc} \\
\text{(mapping } (((x \ y) (scan-multiple 'list} \\
\text{ '(a b c) } \\
\text{ '(1 2 3))))} \\
\text{(list x y)))}
\]

**Nested loops**

**loop**

\[
\text{(loop for x from 1 to 3} \\
\text{ collect (loop for y from 1 to x} \\
\text{ collect y))} \\
\text{�建 (1) (1 2) (1 2 3))}
\]

**iterate**

\[
\text{(iter outer} \\
\text{ (for i below 2} \\
\text{ (iter (for j below 3} \\
\text{ (in outer (collect (list i j)))))} \\
\text{健 (0 0) (0 1) (0 2) (1 0) (1 1) (1 2))}
\]
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]}

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))

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))

Series

with :by

lisp (iterate (((scan-range :from 1 :upto 10 :by 2))) (print i))

Loop and conditionals

loop

with if, else and finally:

;; https://riptutorial.com/common-lisp/example/11095/conditionally-e
(loop repeat 10
   for x = (random 100)
   if (evenp x)
      collect x into evens
   else
      collect x into odds
   finally (return (values evens odds)))
Combining multiple clauses in an if body requires special syntax (and do, and count):

```lisp
(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 straightforward:

```
(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. \texttt{split} sorts one series into multiple according to provided boolean series.

\begin{verbatim}
(\texttt{let* } ((\texttt{number } (\#M(\texttt{lambda } (n) (\texttt{random } 100))))
  (\texttt{scan-range :below } 10)))
  (\texttt{parity } (\#M\texttt{evenp } \texttt{number})))
(\texttt{iterate } ((\texttt{n } \texttt{number}) (p \texttt{parity}))
  (\texttt{when } p (\texttt{format } t "~a is even!~%" n)))
(\texttt{multiple-value-bind } (\texttt{evens } \texttt{odds}) (\texttt{split } \texttt{number } \texttt{parity})
  (\texttt{values } (\texttt{collect } \texttt{evens})
   (\texttt{collect } \texttt{odds})
   (\texttt{collect-length } \texttt{odds})))
\end{verbatim}

Note that although \texttt{iterate} and the three \texttt{collect} expressions are written sequentially, only one iteration is performed, the same as the example with loop.

**Terminate the loop with a test (until, while)**

\texttt{loop}

\begin{verbatim}
(\texttt{loop } \texttt{for } x \texttt{in } '(1 2 3 4 5)
  \texttt{until } (> x 3)
  \texttt{collect } x)
\end{verbatim}

the same, with \texttt{while}:

\begin{verbatim}
(\texttt{loop } \texttt{for } x \texttt{in } '(1 2 3 4 5)
  \texttt{while } (< x 4)
  \texttt{collect } x)
\end{verbatim}

**Series**

We truncate the series with \texttt{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)))

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

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 for i from 1 to 3 count (oddp i))
;; 2

Series

(collect-length (choose-if #'oddp (scan-range :from 1 :upto 3)))

Summation

(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 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)))
```
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)))
```

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” which 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:
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.

**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
- 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
- riptutorial.com

Iterate

- The Iterate Manual - iterate - highlights at a glance and examples
- Loop v Iterate - SabraOnTheHill

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 defines functions generate, permute, displace, flatten, split, combine, reshape. It also defines each, for element-wise operations. This library is not maintained by the original author, but there is an actively maintained fork.
- 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](https://github.com/). 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 code](https://github.com/).

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 2) :initial-element 1.0))
*MY-ARRAY*
* *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.

```lisp
* (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 [github repository](https://github.com) 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:

```lisp
* (aops:generate (lambda () (random 1.0)) '3 3)
#2A((0.99292254 0.929777 0.93538976)
  (0.31522608 0.45167792 0.94118556)
  (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:

```lisp
* (ql:quickload :alexandria)
To **load** "alexandria":
```
Load 1 ASDF system:
alexandria
; Loading "alexandria"

(:ALEXANDRIA)

* (aops:generate #'alexandria:gaussian-random 4)
#(0.5522547885338768d0 -1.2564808468164517d0 0.9488161476129733d0
 -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.

```
(lisp * (defparameter a (1 2 3 4)) A * (aref a 0) 1 * (aref a 3) 4 * (defparameter b (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:

```
(lisp * (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
```
lisp * (defparameter a (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 symbo

  ;; Generate gensyms for dimension sizes
  (let* ((rank (length syms))
         (syms-rev (reverse syms)) ; Reverse, since starting with in
         (dims-rev (loop for i from 0 below rank collecting (gensym)
                        (result `(progn ,@body))) ; Start with innermost expression
         ;; 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 arg
                                           (setf result
                                               `(loop for ,sym from 0 below ,dim do
                                                     result))))
         ;; Add checking of rank and dimension types, and get dimensions
         (let ((dims (gensym)))
           `(let ((,dims ,dimensions))
              (unless (= (length ,dims) ,rank) (error "Incorrect number o
                                                     (dolist (dim ,dims)
                                                       (unless (integerp dim) (error "Dimensions must be integer
                                                                       (destructuring-bind ,((reverse dims-rev) ,dims ; Dimensions
                                                                       ,result)))))))))

so that the contents of a 2D array can be printed using .lisp * (defparameter a #2A((1 2 3) (4 5 6))) A * (nested-loop (i j) (array-dimensions a) (format t “a[~a
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.

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)
```
* (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: ~lisp (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)
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:

```lisp
(defun 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 array
  ;; of the same type for the result
  (let ((size (array-total-size ,,(first variables)))) ;; Total arr
    (result (make-array (array-dimensions ,,(first variables))
      :element-type (array-element-type ,,(first variables))

    ;; Check that all variables have the same size
    ,(mapcar (lambda (var) `(if (not (equal (array-dimensions ,,(array-dimensions var) ,v
            (error "~S and ~S have different

    (rest variables)))

  (doitmes (indx size))
    ;; Locally redefine variables to be scalars at a given inde
    (let ,(mapcar (lambda (var) (list var `((row-major-aref ,var
      ;; User-supplied function body now evaluated for each ind
      (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:

```lisp
(lisp * (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:

\[
\text{\texttt{\*(defparameter a \#(1 2 3 4))}} \\
\text{A} \\
\text{\texttt{\*(defparameter b \#(2 3 4 5))}} \\
\text{B} \\
\text{\texttt{\*(vectorize (a b) \*(a (sin b)))}} \\
\text{#(0.9092974 0.28224 -2.2704074 -3.8356972)}
\]

and combined with \texttt{cmu-infix}:

\[
\text{\texttt{lisp \*(vectorize (a b) \#(a * sin(b))) \#(0.9092974 0.28224 -2.2704074 -3.8356972)}}
\]

### Calling BLAS

Several packages provide wrappers around BLAS, for fast matrix manipulation.

The \texttt{lla} package in quicklisp includes calls to some functions:

#### Scale an array

scaling by a constant factor:

\[
\text{\texttt{\*(defparameter a \#(1 2 3))}} \\
\text{\texttt{lla:scal! 2.0 a}} \\
\text{\texttt{a \#(2.0d0 4.0d0 6.0d0)}}
\]

#### AXPY

This calculates \(a \times x + y\) where \(a\) is a constant, \(x\) and \(y\) are arrays. The \texttt{lla:axpy!} function is destructive, modifying the last argument \((y)\).

\[
\text{\texttt{\*(defparameter x \#(1 2 3))}} \\
\text{A} \\
\text{\texttt{\*(defparameter y \#(2 3 4))}} \\
\text{B} \\
\text{\texttt{lla:axpy! 0.5 x y}}
\]
If the \( y \) array is complex, then this operation calls the complex number versions of these operators:

\[
\begin{align*}
\text{defparameter } & x \ (#(1 2 3)) \\
\text{defparameter } & y \ (\text{make-array } 3 : \text{element-type} \ (\text{complex double-float} \\
& \text{initial-element} \ #C(1d0 1d0))) \\
\text{y} \ (#(#C(1.0d0 1.0d0) #C(1.0d0 1.0d0) #C(1.0d0 1.0d0))) \\
\text{lla:axpy! } \#C(0.5 0.5) a \ b \\
\#(#C(1.5d0 1.5d0) #C(2.0d0 2.0d0) #C(2.5d0 2.5d0))
\end{align*}
\]

**Dot product**

The dot product of two vectors:

\[
\begin{align*}
\text{defparameter } & x \ (#(1 2 3)) \\
\text{defparameter } & y \ (#(2 3 4)) \\
\text{lla:dot } x \ y \\
20.0d0
\end{align*}
\]

**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:

\[
\begin{align*}
\text{defparameter } & a \ #2A((1 2) (3 4))) \\
\text{A} \\
\text{reduce } #\text{max} (\text{make-array} (\text{array-total-size} a) : \text{displaced-to} a)) \\
4
\end{align*}
\]
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 vectorized e 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 variables)
   (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-dimensions ,v
     (array-dimensions ,var)))
       (error "~S and ~S have different dimensions" ,v
     result (gensym))); Returned value
    (rest variables)))

;; Apply FN with the first two elements (or fewer if size < 2
(let ((,result (apply ,fn (loop for ,indx below (min ,size 2)
     (let ,(map 'list (lambda (var) (progn ,@body)))))))))

;; Loop over the remaining indices
(loop for ,indx from 2 below ,size do
    ;; Locally redefine variables to be scalars at a given i
    (let ,(mapcar (lambda (var) (list var `(row-major-aref
     ;; User-supplied function body now evaluated for each
     (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
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:

```lisp
* (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:

```lisp
* (lla:mm #(1 2 3) #(2 3 4))
20
```

**Matrix-vector product**

```lisp
* (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 $A[i,j] \times [j]$

**Matrix-matrix multiply**

* (lla:mm #2A((1 2 3) (1 2 3) (1 2 3)) #2A((2 3 4) (2 3 4) (2 3 4))

which summed over j in $A[i,j] \times B[j,k]$

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 $A[i,j] = B[i] \times C[j]$. 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 #(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 0.24292013188045855d0
  (-0.3744175632000917d0 -0.5810891192666799d0 0.72259734557d0
  (-0.6618248071322363d0 -0.3783451320875919d0 -0.6471807210d0
  :D #S(CL-NUM-UTILS.MATRIX:DIAGONAL-MATRIX
The diagonal matrix (singular values) and vectors can be accessed with functions:

(lla:svd-u a-svd)
#2A((-0.6494608633564334d0 0.7205486773948702d0 0.24292013188045855d0
 -0.3744175632000917d0 -0.5810891192666799d0 0.7225973455785591d0
 -0.6618248071322363d0 -0.3783451320875919d0 -0.6471807210432038d0)

* (lla:svd-d a-svd)
#S(CL-NUM-UTILS.MATRIX:DIAGONAL-MATRIX
 :ELEMENTS #(5.593122609997059d0 1.2364443401235103d0 0.43380279311714376d0))

* (lla:svd-vt a-svd)
#2A((-0.2344460799312531d0 -0.7211054639318696d0 -0.651952410927642134809678d0
 -0.6924017945853318d0 0.6663192365460215d0
 -0.9318994611765425d0 -0.02422116311440764d0 0.3619070730398283d0)

**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))
  #=> (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)))
  #=> (SIMPLE-DENSE-TENSOR (COMPLEX DOUBLE-FLOAT)) [2 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)))
  #=> (SIMPLE-DENSE-TENSOR (COMPLEX DOUBLE-FLOAT)) [3 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
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]`
  ```
  1.000  2.000  3.000
  >
  ```
* `#d[[1,2,3],[4,5,6]]`
  ```
  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))`
  ```
  1.000  2.000  3.000
  >
  ```
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:

```lisp
(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:

```lisp
* (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

```lisp
* (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:

```lisp
* (m:copy (m:ones '(2 3)) 'cons)
```
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>|
     #(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>|
     #(2.000 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>|
     #((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.

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
```

This call 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 \texttt{GET-DECODED-TIME} to get the calendar time representation of the current time directly:

\begin{verbatim}
CL-USER> (get-decoded-time)
\end{verbatim}

is equivalent to

\begin{verbatim}
CL-USER> (decode-universal-time (get-universal-time))
\end{verbatim}

Here is an example of how to use these functions in a program:

\begin{verbatim}
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 tz) (get-decoded-time) (format t "It is now ~2,'0d:~2,'0d:~2,'0d of ~a, ~d/~d, '0 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)
\end{verbatim}

Of course the call to \texttt{GET-DECODED-TIME} above could be replaced by \texttt{(DECODE-UNIVERSAL-TIME n)}, where \texttt{n} is any integer number, to print an arbitrary date. You can also go the other way around: the function \texttt{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:
Note that the result is automatically adjusted for daylight savings time if the time zone is not supplied. If it is supplied, then 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 `<. Another typical problem is how to compute the “temporal distance” between two given dates. Let’s see how to do this with an example: specifically, we will calculate the temporal distance between the first landing on the moon (4:17pm EDT, July 20 1969) and the last takeoff of the space shuttle Challenger (11:38 a.m. EST, January 28, 1986).

```lisp
CL-USER> (setq *moon* (encode-universal-time 0 17 16 20 7 1969 4))
2194805820

CL-USER> (setq *takeoff* (encode-universal-time 0 38 11 28 1 1986 5))
2716303080

CL-USER> (- *takeoff* *moon*)
521497260
```

That’s a bit over 52 million seconds, corresponding to 6035 days, 20 hours and 21 minutes (you can verify this by dividing that number by 60, 60 and 24 in succession). Going beyond days is a bit harder because months and years don’t have a fixed length, but the above is approximately 16 and a half years.

You can in theory use differences between universal times to measure how long the execution of a part of your program took, but the universal times are represented in seconds, and this resolution will usually be too low to be useful. We will see a better method of doing this in the section about internal time.

To sum up, we have seen how to turn a universal time into a calendar time and
vice-versa, how to perform calculations on universal times, and how to format calendar times in a human-readable way. The last piece that is missing is how to parse a string represented as a human-readable string (e.g. “03/11/1997”) and turn it into a calendar time. Unfortunately this turns out to be very difficult in the general case, due to the multitude of different ways of writing dates and times that we use. In some cases it might not even be possible without context information: the above example can be correctly parsed both as March 11th or as November 3rd according to where you are living. In conclusion, either force your users to write dates in a fixed format, or be prepared to write a very intelligent parsing function!

**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:
A good way to see the difference between real time and run time is to test the above code using a call such as (SLEEP 3). The **SLEEP** function suspends the execution of your code for the specified number of seconds. You should therefore see a real time very close to the argument of SLEEP and a run time very close to zero. Let’s turn the above code into a macro in order to make it more general:

```lisp
CL-USER> (defmacro timing (&body forms)
    (let ((real1 (gensym))
          (real2 (gensym))
          (run1 (gensym))
          (run2 (gensym))
          (result (gensym)))
      `(let* ((,real1 (get-internal-real-time))
              ,run1 (get-internal-run-time)
              ,result (progn ,@forms)
              ,run2 (get-internal-run-time)
              ,real2 (get-internal-real-time)))
    (format *debug-io* ";;; Computation took:~%")
    (format *debug-io* ";;; ~f seconds of real time~%" (/ - real2 real1 internal-time-units-per-second))
    (format t ";;; ~f seconds of run time~%" (/ - run2 run1 internal-time-units-per-second)))
TIMING

CL-USER> (timing (sleep 1))
;;; Computation took: 0.994 seconds of real time 0.0 seconds of run
;;; time
NIL
```
The built-in macro **TIME** does roughly the same as the above macro (it executes a form and prints timing information at the end), but it also usually provides information about memory usage, time spent in garbage collection, page faults, etc. The format of the output is implementation-dependent, but in general it’s pretty useful and informative. This is an example under Allegro Common Lisp 6.0: we generate a list of 100 real numbers and we measure the time it takes to sort them in ascending order.

```lisp
CL-USER> (let ((numbers (loop for i from 1 to 100 collect (random 1.
                      (time (sort numbers #'<)))))
       ; cpu time (non-gc) 0 msec user, 10 msec system
       ; cpu time (gc) 0 msec user, 0 msec system
       ; cpu time (total) 0 msec user, 10 msec system
       ; real time 9 msec
       ; space allocation:
       ; 3,586 cons cells, 11,704 other bytes, 0 static bytes

Computing the day of the week

In the section about **Universal Time** we’ve learned enough to write a small function that computes the day of the week. Unfortunately, by definition, this function won’t work for dates before January 1, 1900.

```lisp
CL-USER> (defun day-of-week (day month year)
            "Returns the day of the week as an integer. Monday is 0."
            (nth-value 6
                      (decode-universal-time
                       (encode-universal-time 0 0 0 day month year 0)
                      0)))

DAY-OF-WEEK
CL-USER> (day-of-week 23 12 1965)
3
CL-USER> (day-of-week 1 1 1900)
0
CL-USER> (day-of-week 31 12 1899)

Type-error in KERNEL::OBJECT-NOT-TYPE-ERROR-HANDLER:
1899 is **not** of type (OR (MOD 100) (INTEGER 1900))


If this is a problem for you, here’s a small function by Gerald Doussot (adapted from the comp.lang.c FAQ) that will help you:

```lisp
(defun day-of-week (day month year)
  "Returns the day of the week as an integer. Sunday is 0. Works for dates after 1752."
  (let ((offset '(0 3 2 5 0 3 5 1 4 6 2 4))
        (when (< month 3)
          (decf year 1))
        (mod
         (truncate (+ year
                     (/ year 4)
                     (/ (- year)
                        100)
                     (/ year 400)
                     (nth (1- month) offset)
                     day
                     -1))
         7)))

The local-time library

The local-time library (available on Quicklisp) is a very handy extension to the somewhat limited functionalities as defined by the standard.

In particular, it can

- print timestamp 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 fro.

For example, here is a function that returns Unix times as a human readable string:

```lisp
(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+))
```
See the [manual](#) for the full details.

**Get today’s date**

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.

**Formatting time strings**

```
(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"
```

`format-timestring` takes a stream argument like `format`:

```
(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*`.

It also accepts a `:format` argument. 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-tim
"Wed, 13 Nov 2019 18:11:38 +0100"

With +asctime-format+:


With +iso-week-date-format+:

(local-time:format-timestring nil (local-time:now) :format local-time "2019-W46-3"

**Defining format strings**

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 '(:year "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 :min :sec :msec, long and short notations (“Monday”, “Mo.”), gmt offset, timezone markers 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 (:year
(:hour 2) #\: (:min 2) #\: (:sec 2) #\space :gmt-offset-hhmm)
"See the RFC 1123 for the details about the possible values of the timezone field."

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 \\
```
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:


It returns the universal time which, in turn, we can ingest with the local-time library:

(local-time:universal-to-timestamp *)
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*

(match '(something #(0 1 2))
  ((list a (vector 0 _ b)))
(values a b))
SOMETHING
2

list* pattern:

(match '(1 2 . 3)
  ((list* _ _ x)
   x))
3

Note that using list would match nothing.

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
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

and, or
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

The syntax is guard + subpattern + a test form, and the body.

If the subpattern is true, the test form is evaluated, and if it is true it is matched against subpattern1.

The above returns nil, since (- x y) == 3 does not satisfies evenp.

Nesting patterns

Patterns can be nested:

returns 4.

See more
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**

**Using 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**

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 "xaabbd")
)
```

will yield the same results as:

```
(ppcre:scan "(a)*b" "xaabbd")
```

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 a several ways to extract matching fragments, among them: the scan-to-strings and register-groups-bind functions.

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})\s.(\d{4})" "Fra
It might be more convenient to use CL-PPCRE with the [CL-INTERPOL](https://www.cl-ppcre.org) 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:

```
(interpol:enable-interpol-syntax)
```
Input/Output

Redirecting the Standard Output of your Program

You do it like this:

```lisp
(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:

```lisp
(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
```
Characters with code > 127 cannot be represented in ASCII:

;*** - Character \u0080 cannot be represented in the character set

CMUCL

:EXTERNAL-FORMAT :DEFAULT (untested) - no unicode, so probably no problems.

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-termina
    (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)))
        finally (return s)))

  finally (return s)))
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 input))
  (loop for pos = (read-sequence buf input-stream)
       while (plusp pos)
       do (write-sequence buf output-stream :end pos))))
Files and Directories

Note: In this chapter, we use mainly namestrings to specify filenames. The issue of pathnames needs to be covered separately.

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

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).

```
$ ln -s /etc/passwd foo

* (probe-file "/etc/passwd")
#p"/etc/passwd"

* (probe-file "foo")
#p"/etc/passwd"

* (probe-file "bar")
NIL
```

For more portability, use uiop:probe-file* or uiop:file-exists-p which will return the file pathname (if it exists).

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.

**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.

**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_>
```

...
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 'character)
      for n-characters = (read-sequence buffer in)
      while (< 0 n-characters)
      do (write-sequence buffer out :start 0 :end n-characters)))

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))
```
If the first argument to `peek-char` is a character, the function will skip all characters until that particular character is found:

```lisp
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):

```lisp
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)))
```

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
1
4
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 the file extension

The file extension is a pathname type in Lisp parlance:

```
(pathname-type "~/foo.org") ;; => "org"
```

Getting file attributes (size, access time,...)
**Osicat** (in Quicklisp) is a lightweight operating system interface for Common Lisp on POSIX-like systems, including Windows. With Osicat we can get and set environment variables, manipulate files and directories, pathnames and a bit more.

Once it is installed, Osicat also defines the osicat-posix system, which permits us to get file attributes.

```lisp
(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

**Listing files and directories**

Some functions below return pathnames, so you might need the following:

```lisp
(namestring #p"/foo/bar/baz.txt")  => "/foo/bar/baz.txt"
(directory-namestring #p"/foo/bar/baz.txt")  => "/foo/bar/
(file-namestring #p"/foo/bar/baz.txt")  => "baz.txt"
```

**Listing files in a directory**
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

See `uiop/filesystem:collect-sub*directories`. It takes as arguments:

- a directory
- a recursep function
- a collectp 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.

Example:

```lisp
(defparameter *dirs* nil "All recursive directories.")

(uiop:collect-sub-directories "~/cl-cookbook"
 (constantly t)
 (constantly t)
 (lambda (it) (push it *dirs*)))
```

With cl-fad:walk-directory, we can also collect files, not only subdirectories:

```lisp
(cl-fad:walk-directory "/
 (lambda (name)
   (format t "/~A~%" name))
 :directories t)
```

Finding files matching a pattern

Below we simply list files of a directory and check that their name contains a given string.

```lisp
(remove-if-not (lambda (it)
   (search "App" (namestring it)))
 (uiop:directory-files "/
```

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" #P

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 that 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:

(\texttt{ignore-errors}
  \texttt{(\textbackslash / 3 0))}
; \texttt{in: IGNORE-ERRORS (\textbackslash / 3 0)}
; \texttt{(/ 3 0)}
;
; caught \texttt{STYLE-WARNING:}
; Lisp error during constant folding:
; arithmetic error \texttt{DIVISION-BY-ZERO} signalled
; Operation was (\textbackslash / 3 0).
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)))
```

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
```
We can also catch all conditions by matching t, like in a cond:

```
(handler-case
  (progn
    (format t "This won't work...~\%")
    (/ 3 0))
  t (c)
    (format t "Got an exception: ~a~\%" c)
    (values 0 c)))
```

This workflow is similar to a try/catch as found in other languages, but we can do more.

**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)))
```

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**.

```lisp
(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:

```lisp
(define-condition my-division-by-zero (error)
  ((dividend :initarg :dividend
              :initform nil
              :reader dividend)) ;; <-- we'll get the dividend with ( :
documentation "Custom error when we encounter a division by zero"
```

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:

```lisp
(make-condition 'my-division-by-zero :dividend 3)
```
Note: here’s a quick reminder on classes, if you are not fully operational on the Common Lisp Object System:

```lisp
(make-condition 'my-division-by-zero :dividend 3)
```

and :reader dividend created a *generic function* that is a “getter” for the dividend of a `my-division-by-zero` object:

```lisp
(make-condition 'my-division-by-zero :dividend 3)
```

```
(dividend *)
```

```lisp
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:

```lisp
(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.~" divide
```
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, whether 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 mes
  (/ 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. < new rest
;; 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-ZERO"
      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))
```

That’s alright but let’s just write more human-friendly “reports”: 
(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 new  
  ;;  ;; 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 to m
(force-output *query-io*) ;; Ensure the user sees what he types.
(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" prompt)
            :output s))))))
      ;; We get a string and we want a number.
      ;; We could also use parse-integer, the parse-number library, e
      (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`:

```lisp
(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))
```

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)
```

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, or with a double colon if the symbol is not exported: package::non-exported-symbol.

Now we can choose to import individual symbols 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))
```
Sometimes, we see (:import-from :ppcre), without an explicit import. This helps people using ASDF’s *package inferred system*.

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

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"))
```
You can also collect all these symbols in a list by writing:

```lisp
(let (symbols)
  (do-external-symbols (s (find-package "PACKAGE"))
    (push s symbols))
  symbols)
```

Or you can do it with `LOOP`.

```lisp
(loop for s being the external-symbols of (find-package "PACKAGE")
  collect s)
```

### Package nickname

#### Nickname Provided by Packages

When defining a package, it is trivial to give it a nickname for better user experience. The following example is a snippet of `PROVE` package:

```lisp
(defpackage prove
  (:nicknames :cl-test-more :test-more)
  (:export :run
    :is
    :ok)
```

Afterwards, a user may use nickname instead of the package name to refer to this package. For example:

```lisp
(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:

```lisp
(defpackage #:iterate
  (:nicknames #:iter))

(defpackage :cl-ppcre
  (:nicknames :ppcre)
```

**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 nicknames.

Many implementations (SBCL, CCL, ECL, Clasp, ABCL, ACL, LispWorks >= 7.2...) support Package Local Nicknames (PLN).

```lisp
(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")
```

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.

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.
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:

```lisp
(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 ope
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:

```lisp
(cl-package-locks:without-package-locks
  (rename-package :alexandria :alex))
```

See also
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*: each *quoted* symbol evaluates to itself, aka it is returned as is:

- (quote foo bar baz) returns (foo bar baz)
- the quote character, ', is a shortcut for quote, a *special operator* (not a function nor a macro, but one of a few special operators forming the core of Lisp).
- so, 'foo evaluates 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:

```lisp
(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):

```lisp
(macroexpand '(setq2 v1 v2 3))
;; (PROGN (SETQ V1 3) (SETQ V2 3))
;; T
```

Yay, our macro expands to the code we wanted!

More interestingly:

```lisp
(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:`
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 \((\text{setq} x y (+ z 3))\). 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 \text{setq} expression, it must suddenly switch to executing the body of the \text{setq} 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 \text{setq} macro needs to do some complex transformation on its \textit{e} argument before plugging it into the result. Suppose this transformation can be written as a Lisp procedure \text{some-computation}. The novice will often write:

\begin{verbatim}
(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!_
\end{verbatim}

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 \texttt{defmacro}. This is the wrong picture. The right picture is that \texttt{defmacro} enables a step into the \textit{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:

\begin{verbatim}
(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!_
\end{verbatim}

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 \texttt{setq2-function} in my example). In most cases, it is easy to do so: In defining a macro, we use all the usual \texttt{lambda-list} syntax, such as \texttt{&optional}, \texttt{&rest}, \texttt{&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:

\begin{verbatim}
(defmacro foo (x &optional y &key (cxt 'null)) ...)
\end{verbatim}

then

\begin{verbatim}
  _If we call it thus ..._  | _The parameters' values are ..._
  --|--
 `(foo a)`                  | `x=a`, `y=nil`, `cxt=null`
 `(foo (+ a 1) (- y 1))`  | `x=(+ a 1)`, `y=(- y 1)`, `cxt=null`
 `(foo a b :cxt (zap zip))` | `x=a`, `y=b`, `cxt=(zap zip)`
\end{verbatim}

Note that the values of the variables are the actual expressions (\texttt{(+ a 1)} and (\texttt{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 \texttt{setq}: (\texttt{setq-reversible \texttt{e}_1 \texttt{e}_2 \texttt{d}}) behaves like (\texttt{setq \texttt{e}_1 \texttt{e}_2}) if \texttt{d=:normal}, and behaves like (\texttt{setq \texttt{e}_2 \texttt{e}_1}) if \texttt{d=:backward}. It could be defined thus:

\begin{verbatim}
(defmacro setq-reversible (e1 e2 direction)
  (case direction
    (:normal (list 'setq e1 e2))
    (:backward (list 'setq e2 e1))
    (t (error "Unknown direction: ~a" direction))))
\end{verbatim}

Here’s how it expands:

\begin{verbatim}
(macroexpand '(setq-reversible x y :normal))
\end{verbatim}
(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)))))

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 the list expression given above, one writes

```
'(progn (setq ,v1 ,e) (setq ,v2 ,e))
```

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))
```

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

```lisp
(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

```lisp
(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 popped 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
\begin{verbatim}
(progn
  (setq x (+ x 2))
  (setq y (+ x 2)))
\end{verbatim}

so that \textit{x} will have value 10 and \textit{y} will have value 12. Indeed, here’s its macroexpansion:

\begin{verbatim}
(macroexpand '((setq2 x y (+ x 2)))
 ;;(PROGN (SETQ X (+ X 2)) (SETQ Y (+ X 2)))
\end{verbatim}

Chances are that isn’t what the macro is expected to do (although you never know). Another problematic case is (setq2 x y (pop 1)), which causes 1 to be popped twice; again, probably not right.

The solution is to evaluate the expression \textit{e} just once, save it in a temporary variable, and then set \textit{v1} and \textit{v2} to it.

\textbf{Gensym}

To make temporary variables, we use the \texttt{gensym} function, which returns a fresh variable guaranteed to appear nowhere else. Here is what the macro should look like:

\begin{verbatim}
(defmacro setq2 (v1 v2 e)
  (let (((tempvar (gensym))))
    `(let ((,tempvar ,e))
      (progn (setq ,v1 ,tempvar)
           (setq ,v2 ,tempvar))))))
\end{verbatim}

Now (setq2 x y (+ x 2)) expands to

\begin{verbatim}
(let ((#:g2003 (+ x 2)))
  (progn (setq x #:g2003) (setq y #:g2003)))
\end{verbatim}

Here \texttt{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 \((setq2 \ x \ y \ (\text{pop} \ 11))\).

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 `defmethods` 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

```lisp
(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 (declare (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 11), 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)
        (dolist (y1 y)
          (dolist (z1 z)
            ...
         ))))
  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)
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 :ins 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
Another would be to write

\[
\begin{align*}
\text{(and \symbolp \text{(second listspec))} \\
\text{(eq (intern \symbol-name (second listspec)) :keyword)} \\
\text{'(in))}
\end{align*}
\]

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:

\[
\text{(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, \((\text{build-symbol} (:< x) "-" (:++ *x-num*))\), when \( x = \text{foo} \) and \(*x-num* = 8\), sets \(*x-num* to 9 and evaluates to F00-9. If evaluated again, the result will be F00-10, and so forth.

Obviously, \texttt{build-symbol} can’t be implemented as a function; it has to be a macro. Here is an implementation:

\[
\begin{align*}
\text{(defmacro build-symbol (&rest list) } & \text{ \{)))
\text{(let ((p (find-if (lambda (x) }
\text{\{and (consp x) }
\text{\{eq (car x) ':package))))}
\text{\{list))))
\text{(when p}
\text{\{setq list (remove p list}))}
\text{(let ((pkg (cond ((eq (second p) 'nil) nil)
\text{\{t `(find-package ',(second p))```))))}
\text{\{cond (p}
\text{\{cond (pkg}
\text{\{values (intern ,(symstuff list) ,pkg))}
\text{\{t}
\text{\{make-symbol ,(symstuff list))```))))}
\text{\{t}
\text{\{``(values (intern ,(symstuff list))))))))
\text{\{defun symstuff (list) }
\text{\{`(concatenate 'string }
\text{\{,@(for (x :in list)
\text{\{cond ((stringp x)
\text{\{
\text{\{\text{\{atom x)
\text{\{`,(format nil "~a" x))
\text{\{\text{\{eq (car x) ':<)
\text{\{`(format nil "~a" ,(second x)))
\text{\{\text{\{eq (car x) ':++)
\text{\{`(format nil "~a" (incf ,(second x))))
\text{\{t}
\text{\{`(format nil "~a" ,x))])))
\end{align*}
\]
(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
       ,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 ()))

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. \texttt{(fi s \[k\]):} Find the \(k\)'th subexpression of the current form whose \texttt{car} is \texttt{s}. (\(k\) defaults to 0.) Make that subexpression the current form and pretty-print it.

Suppose you’re trying to debug a macro \texttt{hair-squared} that expands into something complex containing a subform that is itself a macro form beginning with the symbol \texttt{odd-define}. You suspect there is a bug in the subform. You might issue the following commands:

\begin{verbatim}
(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) ...) ...

\end{verbatim}

Once again, it is clear that \texttt{ex} and \texttt{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 \texttt{(command ’x ...)} instead of \texttt{(command x ...)}, because we want to think of the command as a variant of \texttt{setq}.

Here is how \texttt{ex} and \texttt{fi} might be defined:

\begin{verbatim}
(defvar *current-form*)

(defmacro ex (&optional (form nil form-supplied))
  `(progn
    (pprint (setq *current-form*

\end{verbatim}
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](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](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.

```lisp
(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 slots options like this:

```
(defclass point ()
  (x y z))
```
or even without slots 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 slot
    ;; #<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).

```lisp
(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.

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 ((declare (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:

```lisp
(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`.

```lisp
(type-of c1)
;; CHILD
```

```lisp
(subtypep (type-of c1) 'person)
;; T
```

```lisp
(ql:quickload "closer-mop")
;; ...
```

```lisp
(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.

```lisp
(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:

```lisp
(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 ()
  ((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
    :initarg :arg
    :initform 18
    :accessor age)))

(age p1)
;; => slot unbound error. This is different from "slot missing":

(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 func

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

#<PERSON me lisper: t>

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 not CLOS instances:

\[
\begin{align*}
\text{(find-class 'symbol)} \\
;; \text{#<BUILT-IN-CLASS SYMBOL>}
\text{(class-name *)} \\
;; \text{SYMBOL}
\text{(eq ** (class-of 'symbol))} \\
;; \text{T}
\text{(class-of ***)} \\
;; \text{#<STANDARD-CLASS BUILT-IN-CLASS>}
\end{align*}
\]

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.

\[
\begin{align*}
\text{(find-class t)} \\
;; \text{#<BUILT-IN-CLASS T 20305AEC>}
\end{align*}
\]

In addition to classes corresponding to lisp types, there is also a CLOS class for
every structure type you define:

```
(defstruct foo)
F00
```

```
(class-of (make-foo))
;; #<STRUCTURE-CLASS F00 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:

- `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`
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:

```
(defclass example ()
  ((slot1
```
(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 risks©.

Methods

Diving in

Recalling our person and child classes from the beginning:

(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 common-li
(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 DEFGENERIC
;; #<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

(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--&")
)

(greet p1)

(greet c1)

;; Adding in &key

;; In order to add "&key" to our generic method, we need to remove it
(fmakunbound 'greet) ;; with Slime: C-c C-u (slime-undefine-function)

(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))
    (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 (ignoreable meal-type))
    (format t "eating~&")))

(defmethod feed (obj (meal-type eql :dessert))
  (declare (ignoreable meal-type))
  (format t "mmh, dessert !~&"))

(feed c1 :dessert)
;; mmh, dessert !

(defmethod feed ((obj child) (meal-type eql :soup)))
(declare (ignoreable 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 \texttt{foo} is bound to the corresponding argument value, as usual.

2- a variable and a \textbf{specializer}, as in:

\[
\begin{array}{l}
\text{(defmethod greet ((foo person))}
\end{array}
\]

\[
\begin{array}{l}
\text{...)}
\end{array}
\]

In this case, the variable \texttt{foo} is bound to the corresponding argument only if that argument is of specializer class \texttt{person} or a \textit{subclass}, like \texttt{child} (indeed, a “child” is also a “person”).

If any argument fails to match its specializer then the method is not \textit{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”.

\textbf{Only required parameters can be specialized.} We can’t specialize on optional &\texttt{key} arguments.

3- a variable and an \textbf{eql specializer}

\[
\begin{array}{l}
\text{(defmethod feed ((obj child) (meal-type (eql :soup)))}
\end{array}
\]

\[
\begin{array}{l}
\text{(declare (ignorable meal-type))}
\end{array}
\]

\[
\begin{array}{l}
\text{(format t "bwark~&")}
\end{array}
\]

\[
\begin{array}{l}
\text{(feed c1 :soup)}
\end{array}
\]

\[
\begin{array}{l}
\text{;; "bwark"}
\end{array}
\]

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 \textit{congruent} with the shape of the generic function. The system chooses the most \textit{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.")))

(defmethod hug ((a person) (b person))
    :person-person-hug)

(defmethod hug ((a person) (b child))
    :person-child-hug)
```

Read more on [Practical Common Lisp](https://practicalcmlisp.com).  

**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:

```lisp
(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:

```lisp
(defmethod greet ((obj child))
  (format t "ur so cute~&")
  (when (next-method-p)
    (call-next-method)))
;; STYLE-WARNING: REDEFINING GREET (#<STANDARD-CLASS CHILD>) in DEFM
;; #<STANDARD-METHOD GREET (child) {1003D3DB43}>

(greet c1)
```
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 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
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](https://example.com/post).

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

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>

(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-classes

(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-class)
  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) {100771847`
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}>

(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.

```
(defmethod initialize-instance :after ((obj person) &key) ;; note &k
  (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:

```
(defmethod initialize-instance :after ((obj person) &key)
(with-slots (name) obj
  (assert (>= (length name) 3))))

So this call doesn't work anymore:

(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”:

(defmethod INITIALIZE-INSTANCE :after ((obj person) &key)
  (with-slots (name) obj
    (assert (>= (length name) 3)
      (name) ;; creates a restart that offers to change "name
      "The value of name is ~a. It should be longer than 3 cha

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.   < new re
   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

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. 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:

```lisp
(defvar *var* 1234)
*VAR*
(type-of *var*)
(INTEGER 0 461686018427387903)
```

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:

```lisp
(setf *var* "hello")
"hello"
(type-of *var*)
(SIMPLE-ARRAY CHARACTER (5))
```
You see, type-of returns a different result: \textit{simple-array} of length 5 with contents of type \texttt{character}. This is because \texttt{*var*} is evaluated to string "hello" and the function type-of actually returns the type of object "hello" instead of variable \texttt{*var*}.

**Type Hierarchy**

The inheritance relationship of Lisp types consists a type graph and the root of all types is \texttt{T}. For example:

\begin{verbatim}
* (describe 'integer)
COMMON-LISP:INTEGER
    [symbol]

INTEGER names the \texttt{built-in-class} #<BUILT-IN-CLASS COMMON-LISP:INTEGER>
    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 '*'))
\end{verbatim}

The function \texttt{describe} shows that the symbol \texttt{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 \texttt{type specifiers}.

There are some differences between the function \texttt{type-of} and \texttt{class-of}. The function \texttt{type-of} returns the type of a given object in type specifier format while \texttt{class-of} returns the implementation details.

\begin{verbatim}
* (type-of 1234)
(INTEGER 0 4611686018427387903)
\end{verbatim}
* (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)
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))
As you may have guessed, the type specifier above can be shortened as following:

* (typep '#(1 2 3) 'vector)

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**).

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

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 datos

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 and declare. 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.

```lisp
(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 ;)

**Declaring the type of variables**

Use the macro **declaim**.

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:

(declaim (type (list-of-strings) *all-names*))

;; and with a wrong value:
(defparameter *all-names* "")

;; we get an error:
Cannot set SYMBOL-VALUE of *ALL- NAMES* to ", not of type
(SATISFIES LIST-OF-STRINGS-P).
[Condition of type SIMPLE-TYPE-ERROR]

We can compose types:

(declaim (type (or null list-of-strings) *all-names*))

 Declaring the input and output types of functions

We use again the declare macro, with ftype (function ...) instead of just type:

(declaim (ftype (function (fixnum) fixnum) add))

(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:

(defun add (n)
  (format nil "~a" (+ n 1)))

; caught WARNING:
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:

```lisp
(declaim (ftype (function (string &key (:n integer))) foo))
(defun foo (bar &key n) …)
```
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:

```lisp
(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:

```lisp
(-> 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:

```lisp
(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, 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))) is
; (VALUES (OR NULL NUMBER) &OPTIONAL),
; conflicting with the declared function return type
; (VALUES STRING &REST T).
```

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 (pre-alpha): adding Hindley-Milner type checking to Common Lisp which allows for gradual adoption, in the same way Typed Racket or Hack allows for. It is as an embedded DSL in Lisp that resembles Standard ML or OCaml, but lets you seamlessly interoperate with non-statically-typed Lisp code (and vice versa).

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
    (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)

Voila! 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 https://code.google.com/p/blackthorn-engine-3d/source/browse/src/examples/usocket/usocket.lisp 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))

Voila! 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 https://gist.github.com/shortsightedsid/71cf34282dfae0dd2528
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** is a command-line argument parser, 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 NAME." (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)
      #+#gcl (si:getenv name)
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*
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/cbro
```

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 `unix-opts`.

```
(ql:quickload "unix-opts")
```

We can now refer to it with its opts nickname.

We first declare our arguments with `opts:define-opts`, for example

```
(opts:define-opts
  (:name :help
    :description "print this help text"
  :short #\h
```
Everything should be self-explanatory. Note that `parse-integer` is a built-in CL function.

Now we can parse and get them with `opts:get-opts`, which returns two values: the first is the list of valid options and the second the remaining free arguments. We then must use `multiple-value-bind` to catch everything:

```
(multiple-value-bind (options free-args)
    (opts:get-opts '("hello" "-h" "-l" "1"))
    (format t "Options: ~a~&" options)
    (format t "free args: ~a~&" free-args))
```

Options: (HELP T LEVEL 1)
free args: (hello)
NIL

If we put an unknown option, we get into the debugger. We refer you to unix-opts’ documentation and code sample to deal with erroneous options and other errors.

We can access the arguments stored in `options` with `getf` (it is a property list), and we can exit (in a portable way) with `opts:exit`. So, for example:

```
(multiple-value-bind (options free-args)
    ;; No error handling.
    (opts:get-opts)
```
(if (getf options :help)
 (progn
   (opts:describe
     :prefix "My app. Usage:"
     :args "[keywords]"
   (exit))) ;; <= exit takes an optional return status.
...

And that’s it for now, you know the essential. See the documentation for a complete example, and the Awesome CL list for useful packages to use in the terminal (ansi colors, printing tables and progress bars, interfaces to readline,…).

**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:

```lisp
(uiop:run-program "firefox")
```

or

```lisp
(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:

```lisp
run-program (command &rest keys &key
```
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 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 #+clozure *default-stream-element-type*
    #+clozure 'character)
  (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 the
  ;; 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 (zerop
  ;; exit-code) <-> success, an exit in response to a signal is
  ;; encoded as 128+signum.
  (exit-code :initform nil)
  ;; If the platform allows it, distinguish exiting with a code
  ;; >128 from exiting in response to a signal by setting this co
  (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 :
  ;; 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
An exit code to 0 means success (use \texttt{zerop}).

The exit code is also stored in the \texttt{exit-code} slot of our \texttt{process-info} object. We see from the class definition above that it has no accessor, so we’ll use \texttt{slot-value}. It has an \texttt{initform} to nil, so we don’t have to check if the slot is bound.

We can do:

\begin{verbatim}
(slot-value *my-process* 'uiop/launch-program::exit-code)
\end{verbatim}

The trick is that we \textit{must} run \texttt{wait-process} beforehand, otherwise the result will be nil.

Since \texttt{wait-process} is blocking, we can do it on a new thread:

\begin{verbatim}
(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>"
)
\end{verbatim}

Note that \texttt{run-program} returns the exit code as the third value.

\textbf{Input and output from subprocess}

If the input keyword is set to :\texttt{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
`; 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
    (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.

**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)
[...]
```
Threads

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, [SBCL threads](#) and the [lparallel](#) library.

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.

**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 spawning 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*)
```

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

```lisp
;;; Print the current thread, all the threads, and the current t
(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~%~a" curr-thread)
    (format t "Current thread name: ~a~%~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: #<WAITQUEUE {1003A290063}>
  #<THREAD "reader-thread" RUNNING {1003A200063}>
  #<THREAD "control-thread" waiting on: #<WAITQUEUE {1003A19E53}>
  #<THREAD "Swank Sentinel" waiting on: #<WAITQUEUE {1003790043}>
```
Update a global variable from a thread:

```lisp
(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 — fixed
(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:
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:

```lisp
;;;; Print a message onto the top-level using a thread - reader macro

(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:

```lisp
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 {10
  ==>
    #<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))

(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:

(defparameter *rich*
  (make-instance 'bank-account
    :id 1
    :name "Rich"
    :balance 0))

(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> (dотimes 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:

```lisp
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:

```lisp
(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 thread")
  (if (bt:thread-alive-p destroyer-thread)
      (progn
        (format t "~%[Main Thread] Destroyer thread alive... k
                 (bt:destroy-thread destroyer-thread))
        (format t "~%[Main Thread] Destroyer thread is already d
                 (until (bt:thread-alive-p destroyer-thread))
        (format t "[Main Thread] Waiting for destroyer thread
                 (sleep (* 0.01 (random 100000))))
    )
```

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.

**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)

**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:

```clojure
;;; Print the current thread, all the threads, and the current t
(defun print-thread-info ()
  (let* ((curr-thread sb-thread:*current-thread*)
```
(curr-thread-name (sb-thread:thread-name curr-thread))
(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)

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: #<WAITQUEUE>
  #<THREAD "reader-thread" RUNNING {1003A20063}>
  #<THREAD "control-thread" waiting on: #<WAITQUEUE {1003A19E53}>
  #<THREAD "Swank Sentinel" waiting on: #<WAITQUEUE {1003790043}>
  #<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:

```lisp
;;; 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:

```lisp
;;; Print a message onto the top-level using a thread - fixed

(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:
Print a message onto the top-level — better

The code:

;;; Print a message onto the top-level using a thread - reader macro
(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))

(defun deposit (account amount)
  (:documentation "Deposit money into the account"))

(defun withdraw (account amount)
  (:documentation "Withdraw amount from account"))

(defun method deposit ((account bank-account) (amount real))
  (incf (:balance account) amount))

(defun method withdraw ((account bank-account) (amount real))
  (decf (:balance account) amount))

(defun parameter *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 real))
  (sb-ext:atomic-incf (car (cons (:balance account) nil)) amount)
(defmethod atomic-withdraw ((account bank-account) (amount real) (sb-ext:atomic-decf (car (cons (:balance account) nil))) amount)

(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
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:

```lisp
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, 23.4368 747.61% CPU
126,674,011,941 processor cycles
3,329,504 bytes consed
NIL
```

With atomics:

```lisp
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, 2.051195 system) run times consist of 0.420 seconds GC time, and 7.756 second 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:

```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 100000))))))))

  (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... killing it" (sb-thread:terminate-thread destroyer-thread))
    (format t "~%[Main Thread] Destroyer thread is already dead")
    (until (sb-thread:thread-alive-p destroyer-thread)
      (format t "~%[Main Thread] Waiting for destroyer thread")
      (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](#)
- [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 \texttt{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 \texttt{channels}, \texttt{promises}, \texttt{futures}, and \texttt{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 \texttt{lparallel:pmap} function is the parallel equivalent of the Common Lisp \texttt{map} function.
- Finally, close the kernel created in the first step using \texttt{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.

\textit{Credit: this article first appeared on z0ltan.wordpress.com.}

\textbf{Installation}

Let’s check if lparallel is available for download using Quicklisp:

\begin{verbatim}
CL-USER> (ql:system-apropos "lparallel")
#<SYSTEM lparallel / lparallel-20160825-git / quicklisp 2016-08-25>
#<SYSTEM lparallel-bench / lparallel-20160825-git / quicklisp 2016-0
#<SYSTEM lparallel-test / lparallel-20160825-git / quicklisp 2016-08
; No value
\end{verbatim}

Looks like it is. Let’s go ahead and install it:
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" :worker-count 8 :use)

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 {10044
  *ERROR-OUTPUT* . #<SLIME-OUTPUT-STREAM {10044EEE
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 {100723FA8
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {100723FE2
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {10072581E
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {100725858
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {100725892
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {1007258CC
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {100725906
 #<SB-THREAD:THREAD "custom--kernel" FINISHED values: NIL {100725940

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-multiple-ta
`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:

```lisp
(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 argum 'initial-contents'.

By default there is no limit on the queue capacity. Passing 'fixed-capacity' keyword argument limits the capacity to the passed. 'push-queue' will block for a full fixed-capacity qu
Source file: /Users/z0ltan/quicklisp/dists/quicklisp/software/

MAKE-QUEUE has a compiler-macro:
 Source file: /Users/z0ltan/quicklisp/dists/quicklisp/software/
; No value
So, as we can see, it supports the following keyword arguments: \textit{:fixed-capacity}, and \textit{initial-contents}.

Now, if we do specify \textit{:fixed-capacity}, then the actual type of the queue will be \texttt{lparallel.vector-queue}, and if we skip that keyword argument, the queue will be of type \texttt{lparallel.cons-queue} (which is a queue of unlimited size), as can be seen from the output of the following snippet:

\begin{verbatim}
(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-one))
    (format t "queue-two is of type: ~a-" (type-of queue-two)))

LPARALLEL-USER> (check-queue-types)
queue-one is of type: VECTOR-QUEUE
queue-two is of type: CONS-QUEUE
NIL
\end{verbatim}

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!

\begin{verbatim}
(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)))
\end{verbatim}
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 7396 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).

```lisp
;;; 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)))
                     (sleep (random 2))
                     (format stream "~d~%" (* x))))) (rand
    (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 = ~
(let ((*task-category* 'cubing-tasks))
(dotimes (i 5)
(submit-task channel #'(lambda (x)
(sleep (random 5)))
(format stream "~%[Cubing] ~d = ~d"
(sleep 1)
(if (evenp (random 10)))
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:

```lisp
(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
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)))
    (doitimes (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
```
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:

```lisp
;;;; 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
                         (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 parallel 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:

```lisp
(defvar *chars*
  (remove-duplicates
   (sort
    (loop for c across "The quick brown fox jumps over the lazy dog"
      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:map, lparallel:pmapc, lparallel:pmapcar, etc.) take two special keyword arguments - :size, specifying the number of elements of the input sequence(s) to process, and - :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)

lpmap: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> (dotoimes (i 10)
  (print (test-por)))

300
300
100
100
100
300
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.

```lisp
;;; 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.**

```lisp
;;; pfuncall - macro
(defun test-pfuncall ()
  (pfuncall #'* 1 2 3 4 5))
```

Sample run:
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.

```lisp
;;; 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.

```lisp
;;; 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 20 22 24 26 30 32 34 36 38 40 42 44 46 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96

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.

;;; pcound - function
(defun test-pcount ()
  (let ((chars "The quick brown fox jumps over the lazy dog"))
    (pcount #'\e chars)))

Sample run:

LPARALLEL-USER> (test-pcount)
parallel:psort: parallel version of sort.

;;; psort - function
(defstruct person
  name
  age)

(defun test-psort ()
  (let* ((names (list "Rich" "Peter" "Sybil" "Basil" "Candy" "Slava"
                    "Olga")
          (people (loop for name in names
                        collect (make-person :name name :age (+ (ran
                          * 100) name))))
  (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 "Rich" :AGE 38) #S(PERSON :NAME "Peter" :AGE 24
  #S(PERSON :NAME "Sybil" :AGE 20) #S(PERSON :NAME "Basil" :AGE 2
  #S(PERSON :NAME "Candy" :AGE 23) #S(PERSON :NAME "Slava" :AGE 3
  #S(PERSON :NAME "Olga" :AGE 33))

"After sorting..."
(#S(PERSON :NAME "Sybil" :AGE 20) #S(PERSON :NAME "Basil" :AGE 2
  #S(PERSON :NAME "Candy" :AGE 23) #S(PERSON :NAME "Peter" :AGE 2
  #S(PERSON :NAME "Olga" :AGE 33) #S(PERSON :NAME "Slava" :AGE 37
  #S(PERSON :NAME "Rich" :AGE 38))
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**


**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](https).

**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 API docs hosted on Quickdoc
- The official homepage of the lparallel library
- 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

The most trivial use of ASDF is by calling (asdf:make "foobar") (or 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:

```
(ql:quickload "foobar")
```

Also, you can use SLIME to load a system, using the M-x slime-load-system Emacs command. 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:make "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:

```
(defsystem "foobar"
  :depends-on ("alexandria" "trivia")
  :components ((:file "foobart")))
```

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)
```
Instead of using multiple complete packages, you might want to just import parts of them:

```lisp
(defun some-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:make "foobar").

If your Lisp was already started when you created that file, you may have to, either:

- load the new .asd file: (load "path/to/foobar.asd"), or with C-c C-k in Slime to compile and load the whole file.
- (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:

```lisp
(defsystem "foobar"
  :depends-on ("alexandria" "trivia")
  :components ( (:file "foobar")
  :in-order-to ((test-op (test-op "foobar/tests"))))

(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

```lisp
(ql:quickload "cl-project")
```

Create a project:

```lisp
(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.markdown
;  writing /Users/fukamachi/Programs/lib/cl-sample/cl-sample-test.asd
;  writing /Users/fukamachi/Programs/lib/cl-sample/cl-sample.asd
;  writing /Users/fukamachi/Programs/lib/cl-sample/src/hogehoge.lisp
;  writing /Users/fukamachi/Programs/lib/cl-sample/t/hogehoge.lisp
;=> 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 readable representation of its argument, which means what is printed can be read back in by the Lisp reader.

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.

Logging

Logging is already a good evolution from print debugging ;)

log4cl is the popular, de-facto logging library but it isn’t the only one. Download it:

(ql:quickload "log4cl")

and let’s have a dummy variable:

(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*)
```

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 nil)) 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. 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)))
We right-click on the #<FOO> object and choose “inspect”. We are presented an interactive pane (in Slime):

```lisp
#<FOO {100F2B6183}>
```

```
Class: #<STANDARD-CLASS FOO>
```

```
Group slots by inheritance [ ]
Sort slots alphabetically [X]
```

```
All Slots:
[ ]  A = (:A :B :C)
[ ]  B = :B
```

```
[set value]  [make unbound]
```

When we click or press enter on the line of slot A, we inspect it further:

```lisp
#<CONS {100F5E2A07}>
```

```
A proper list:
0: :A
1: :B
2: :C
```

**The interactive debugger**

Whenever an exceptional situation happens (see error handling), the interactive debugger pops up.

It presents the error message, 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 on a frame for more details
- we can explore the functionality with the menu that should appear in our
editor. See the “break” section below for a few more commands (eval in frame, etc).

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:

```
(declare (optimize (speed 0) (space 0) (debug 3)))
```

and recompile our code.

**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))
```

```
(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)`.
In Slime we also have the shortcut \texttt{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:

\begin{verbatim}
(declaim (optimize (debug 3)))
\end{verbatim}

The output is printed to \texttt{*trace-output*} (see the CLHS).

In Slime, we also have an interactive trace dialog with \texttt{M-x slime-trace-dialo} bound to \texttt{C-c T}.

\section*{Tracing method invocation}

In SBCL, we can use \texttt{(trace foo :methods t)} to trace the execution order of method combination (before, after, around methods). For example:

\begin{verbatim}
(trace foo :methods t)
\end{verbatim}

\begin{verbatim}
(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
\end{verbatim}

See the \texttt{CLOS} section for a tad more information.

\section*{Step}

\texttt{step} is an interactive command with similar scope than \texttt{trace}. This:
(step (factorial 2))

gives an interactive pane with the available restarts:

Evaluating call:
   (FACTORIAL 2)
With arguments:
   2
   [Condition of type SB-EXT:STEP-FORM-CONDITION]

Restarts:
0: [STEP-CONTINUE] Resume normal execution
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: ((LAMBDA ()))
1: (SB-INT:SIMPLE-EVAL-IN-LEXENV (LET ((SB-IMPL::*STEP-OUT* :MAYBE))
2: (SB-INT:SIMPLE-EVAL-IN-LEXENV (STEP (FACTORIAL 2)) #<NULL-LEXENV
3: (EVAL (STEP (FACTORIAL 2)))

Stepping is useful, however it may be a sign that you need to simplify your function.

**Break**

A call to break makes the program enter the debugger, from which we can inspect the call stack.

**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).

**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.

**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
;; 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 ()
  (bt:make-thread (lambda ()
    (swank:create-server :port 4006)
    :name "swank")
  (format t "we are past go!~%")
  (loop while t do
    (sleep 5)
    (dostuff)
    (incf *counter*))

(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:

#<SB-THREAD:THREAD "Swank 4006" RUNNING {1003A19333}> 

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 \texttt{M-x slime-connect}, choosing localhost for the host and port 4006.

We can write new code:

```lisp
(defun dostuff ()
  (format t "goodbye world ~a~%" *counter*))
(setf *counter* 0)
```

and eval it as usual with \texttt{C-c C-c} or \texttt{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
- cyberrycom: 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
  0.000000 user, 0.000000 system
  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:

![FlameGraph chart]

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:

```lisp
* (defun plus (a b)  
    (+ a b))
PLUS

* (disassemble 'plus)  ; disassembly for PLUS  ; Size: 37 bytes. Origin: #x52B8063B  ; 3B:  498B5D60 MOV RBX, [R13+96]  ; no-a  ; thre  ; 3F:  48895DF8 MOV [RBP-8], RBX
```
The code above was evaluated in SBCL. In some other implementations such as CLISP, disassembly might return something different:

```lisp
* (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.

```lisp
* (defun max-original (a b)
  (max a b))
MAX-ORIGINAL

* (disassemble 'max-original)
; disassemble for MAX-ORIGINAL
; Size: 144 bytes. Origin: #x52D450EF
; 7A7:   8D46F1 lea eax, [rsi-15] ; no-a
; 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
*(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 ; no-a
; 3F: 488945E8  mov [rbp-24], rax
; 43: 488BD0  mov rdx, rax
; 46: 488BF0  mov rdi, rbx
; 49: B9C1030020  mov ecx, 536871873 ; gene
; 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: 488BF8  mov rdi, rbx
; 7E: B929040020  mov ecx, 536871977 ; gene
; 83: FFD1  call rcx
; 85: 488B45E8  mov rax, [rbp-24]
; 89: 488B75F0  mov rsi, [rbp-16]
; 8D: 4C8B45F8  mov r8, [rbp-8]
; 91: 4C0F44C6  cmovneq 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](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 ; no-a  
; 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 ; gene  
; 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 system  
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 system
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 some (like starts of blocks) are unevaluated expressions, but here the expression is 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.
* (declare (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
...

* (declare (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 declare 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 declare 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:

```
(declare (ftype (function (arg1 arg2 ...) return-value) function-name))
```

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 \texttt{square} is a \texttt{fixnum}, the value of the function will also be a \texttt{fixnum}:

\begin{verbatim}
(declaim (ftype (function (fixnum) fixnum) square))
(defun square (x) (* x x))
\end{verbatim}

If we provide it with the argument which is not declared to be of type \texttt{fixnum}, no optimization will take place:

\begin{verbatim}
(defun do-some-arithmetic (x)
 (the fixnum (+ x (square x))))
\end{verbatim}

Now let’s try to optimize the speed. The compiler will state that there is type uncertainty:

\begin{verbatim}
(defun do-some-arithmetic (x)
 (declare (optimize (speed 3) (debug 0) (safety 0)))
 (the fixnum (+ x (square x))))
\end{verbatim}

; compiling (DEFUN DO-SOME-ARITHMETIC ...)
;
; file: /tmp/slimerzRzDh1R
; in: DEFUN DO-SOME-ARITHMETIC
;
; (+ TEST-FRAMEWORK::X (TEST-FRAMEWORK::SQUARE TEST-FRAMEWORK::X
; ;
; ; 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) bec
; ; 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
Now we can add a type declaration for \( x \), so the compiler can assume that the expression \((\text{square } x)\) is a \texttt{fixnum}, and use the \texttt{fixnum}-specific \texttt{+}:

\[
\text{(defun do-some-arithmetic (x)} \text{)}
\text{(declare (optimize (speed 3) (debug 0) (safety 0)}))
\text{(declare (type fixnum x))}
\text{(the fixnum (+ x (square x)}))
\]

; disassemble 'do-some-arithmetic
; Size: 48 bytes. Origin: #x52C084DA
; 4DA: 488945F8 MOV [RBP-8], RAX ; no-a
; 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 ; #<FD
; 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
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.

```lisp
;;; The globally defined function DISPATCH should be open-coded, unless a NOTINLINE declaration overrides this effect.
(declaim (inline dispatch))
(defun dispatch (x) (funcall (get (car x) 'dispatch) x))

;;; 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 system
  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 system
  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*)

When this feature is present, all inlinable generic functions are inlined unless it is declared notinline.

Using a program from a REPL is fine and well, but 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.

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:

```lisp
(sb-ext:save-lisp-and-die #P"path/name-of-executable" :toplevel #'my
```

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.

So we must run the command from a simple SBCL repl.

I suppose your project has Quicklisp dependencies. You must then:

- ensure Quicklisp is installed and loaded at the Lisp startup (you completed Quicklisp installation),
- load the project’s .asd,
- install the dependencies,
- build the executable.

That gives:

```
(load "my-app.asd")
(ql:quickload "my-app")
(sb-ext:save-lisp-and-die #p"my-app-binary" :toplevel #'my-app:main
```

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 #'
```

**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 "<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 Roswell or Buildapp**

Roswell, an implementation manager and much more, also has the ros build command, that should work for many implementations.

We can also make our app installable with Roswell by a ros install my-app. See its documentation.

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 application. 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 webserver on the foreground. In our main function, we can do something like this:

```lisp
(defun main ()
  (start-app :port 9003) ;; our start-app, for example clack:clack-up
  ;; let the webserver run.
  ;; warning: hardcoded "hunchentoot".
  (handler-case (bt:join-thread (find-if (lambda (th)
                                        (search "hunchentoot" (bt:thread-name th))))
                             (bt:all-threads))
    ;; Catch a user's C-c
    (+sbcl sb-sys:interactive-interrupt
      ccl:interrupt-signal-condition
      clisp system::simple-interrupt-condition
      ecl ext:interactive-interrupt
      allegro excl:interactive-signal
      ()
      (progn
        (format *error-output* "Aborting.~&")
        (clack:stop *server*)
        (uiop:quit)))
    (error (c) (format t "Woops, an unknown error occurred:~a" c))
```

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](https://www.reddit.com), 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.

<table>
<thead>
<tr>
<th>program size</th>
<th>implementation</th>
<th>CPU</th>
<th>startup time</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>/bin/true</td>
<td>15%</td>
<td>.0004</td>
</tr>
<tr>
<td>1005</td>
<td>ecl</td>
<td>115%</td>
<td>.5093</td>
</tr>
<tr>
<td>48151</td>
<td>sbcl</td>
<td>91%</td>
<td>.0064</td>
</tr>
<tr>
<td>27054</td>
<td>ccl</td>
<td>93%</td>
<td>.0060</td>
</tr>
<tr>
<td>10162</td>
<td>clisp</td>
<td>96%</td>
<td>.0170</td>
</tr>
<tr>
<td>4901</td>
<td>ecl.big</td>
<td>113%</td>
<td>.8223</td>
</tr>
<tr>
<td>70413</td>
<td>sbcl.big</td>
<td>93%</td>
<td>.0073</td>
</tr>
<tr>
<td>41713</td>
<td>ccl.big</td>
<td>95%</td>
<td>.0094</td>
</tr>
<tr>
<td>19948</td>
<td>clisp.big</td>
<td>97%</td>
<td>.0259</td>
</tr>
</tbody>
</table>

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, we passed from 120MB to 23MB, for a loss of a dozen milliseconds of start-up time, which was still under 50ms!

Your SBCL must be built with core compression, see the documentation: [http://www.sbcl.org/manual/#Saving-a-Core-Image](http://www.sbcl.org/manual/#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 the `:compression` argument may be an integer from -1 to 9, corresponding to zlib compression levels, or `t` (which is equivalent to the default compression level, -1).

We experienced a 1MB difference between levels -1 and 9.

**With ASDF**

However, we prefer to do this with ASDF (or rather, UIOP). Add this in your `.asd`:

```lisp
#+sb-core-compression
(defmethod asdf:perform ((o asdf:image-op) (c asdf:system))
  (uiop:dump-image (asdf:output-file o c) :executable t :compression
```

**With Deploy**

Also, the [Deploy](https://www.debian.org/doc/dpkg/deploy.html) 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.

A quick look at the awesome-cl#scripting list made us choose the unix-opt library.

(ql:quickload "unix-opt")

We can call it with its opts alias (a global nickname).

As often work happens in two phases:

- declaring the options that our application accepts, their optional argument, defining their type (string, integer,…), their long and short names, and the required ones
- parsing them (and handling missing or malformed parameters).

**Declaring arguments**

We define the arguments with opts:define-opts:

```lisp
(opts:define-opts
  (:name :help
    :description "print this help text"
    :short #\h
    :long "help")
  (:name :nb
    :description "here we want a number argument"
    :short #\n
    :long "nb"
    :arg-parser #'parse-integer) ;; <- takes an argument
  (:name :info
    :description "info"
    :short #\i
    :long "info"))
```

Here parse-integer is a built-in CL function. If the argument you expect is a string, you don’t have to define an arg-parser.
Here is an example output on the command line after we build and run a binary of our application. The help message was auto-generated:

$ my-app -h
my-app. Usage:

Available options:
- -h, --help
- -n, --nb ARG
- -i, --info

Usage:

Available options:
- -h, --help          print this help text
- -n, --nb ARG        here we want a number argument
- -i, --info          info

Parsing

We parse and get the arguments with opts:get-opts, which returns two values: the list of valid options and the remaining free arguments. We then must use multiple-value-bind to assign both into variables:

```
(multiple-value-bind (options free-args)
  ;; There is no error handling yet.
  (opts:get-opts)
  ...
```

We can test this by giving a list of strings to get-opts:

```
(multiple-value-bind (options free-args)
  (opts:get-opts '("hello" "-h" "-n" "1"))
  (format t "Options: ~a~&" options)
  (format t "free args: ~a~&" free-args))
Options: (HELP T NB-RESULTS 1)
free args: (hello)
NIL
```

If we put an unknown option, we get into the debugger. We’ll see error handling in a moment.

So options is a property list. We use getf and setf with plists, so that’s how we do our logic. Below we print the help with opts:describe and then we quit (in a portable way).
(multiple-value-bind (options free-args)
  (opts:get-opts)

  (if (getf options :help)
      (progn
        (opts:describe
         :prefix "You're in my-app. Usage:
            :args "[keywords]" ;; to replace "ARG" in "--nb ARG"
        (uiop:quit)))
      (if (getf options :nb)
        ...)

For a full example, see its [official example](#) and [cl-torrents’ tutorial](#).

The example in the unix-opt's repository suggests a macro to do slightly better. Now to error handling.

**Handling malformed or missing arguments**

There are 4 situations that unix-opt’s doesn’t handle, but signals conditions for us to take care of:

- when it sees an unknown argument, an `unknown-option` condition is signaled.
- when an argument is missing, it signals a `missing-arg` condition.
- when it can’t parse an argument, it signals `arg-parser-failed`. For example, if it expected an integer but got text.
- when it doesn’t see a required option, it signals `missing-required-option`.

So, we must create simple functions to handle those conditions, and surround the parsing of the options with a `handler-bind` form:

```
(multiple-value-bind (options free-args)
  (handler-bind (((opts:unknown-option #'unknown-option) ;; the c
                  (opts:missing-arg #'missing-arg)
                  (opts:arg-parser-failed #'arg-parser-failed)
                  (opts:missing-required-option))
    (opts:get-opts))

  ... ;; use "options" and "free-args"
```
Here we suppose we want one function to handle each case, but it could be a simple one. They take the condition as argument.

```lisp
(defun handle-arg-parser-condition (condition)
  (format t "Problem while parsing option ~s: ~a .~%" (opts:option c condition))
  (opts:describe) ;; print help
  (uiop:quit 1))
```

For more about condition handling, see error and condition handling.

**Catching a C-c termination signal**

Let’s build a simple binary, run it, try a C-c and read the stacktrace:

```
$ ./my-app
sleep...
^C
debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT in thread  <= c
#<THREAD "main thread" RUNNING \{1003156A03}\>:
  Interactive interrupt at \#x7FFFF6C6C170.
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.
```

restarted (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE       ] Return from SB-UNIX:SIGINT.  <= i
  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:

```
(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:

```
(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](https://www.example.com).

**Credit**

- cl-torrents’ tutorial
- lisp-journey/web-dev
Testing

So you want to easily test the code you’re writing? The following recipes cover how to write automated tests and see their code coverage. We also give pointers to plug those in modern continuous integration services like Travis CI and Coveralls.

We will use an established and well-designed regression testing framework called Prove. It is not the only possibility though, FiveAM is a popular one (see this blogpost for an introduction) and there are others (and more again). We prefer Prove for its documentation and its extensible reporters (it has different report styles and we can extend them).

warning: Prove has a couple limitations and will soon be obsolete. We advise to start with another test framework.

Testing with Prove

Install and load

Prove is in Quicklisp:

(ql:quickload "prove")

This command installs prove if necessary, and loads it.

Write a test file

(in-package :cl-user)
(defpackage my-test
  (:use :cl
       :prove))
(in-package :my-test)

(subtest "Showing off Prove"
(ok (not (find 4 '(1 2 3))))
(is 4 4)
(isnt 1 #\1))

Prove’s API contains the following testing functions: ok, is, isnt, is-values, is-type, like (for regexps), is-print (checks the standard output), is-error, is-expand, pass, fail, skip, subtest.

**Run a test file**

(prove:run #P"myapp/tests/my-test.lisp")
(prove:run #P"myapp/tests/my-test.lisp" :reporter :list)

We get an output like:

![Test output](image)

**Run one test**
You can directly run one test by compiling it. With Slime, use the usual \texttt{C-c C-c}.

**More about Prove**

Prove can also:

- be run on Travis CI,
- colorize the output,
- report tests duration,
- change the default test function,
- set a threshold for slow tests,
- invoke the CL debugger whenever getting an error during running tests,
- integrate with ASDF so than we can execute \texttt{(asdf:test-system)} or \texttt{(prove:run)} in the REPL (such configuration is provided by \texttt{cl-project}, by the same author).

See [Prove’s documentation](#)!  

**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 \texttt{restart}.

With Prove, set \texttt{prove:*debug-on-error*} to \texttt{t}.

Note that in the debugger:

- \texttt{<enter>} on a backtrace shows more of it
- \texttt{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:

**Coverage report: /tmp/test1.lisp**

<table>
<thead>
<tr>
<th>Kind</th>
<th>Covered</th>
<th>All</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>17</td>
<td>29</td>
<td>58.6</td>
</tr>
<tr>
<td>branch</td>
<td>6</td>
<td>10</td>
<td>60.0</td>
</tr>
</tbody>
</table>

**Key**
- Not instrumented
- Conditionalized out
- Executed
- Not executed
- Both branches taken
- One branch taken
- Neither branch taken

```lisp
(declare (optimize sb-cover:store-coverage-data : quality 3))
(declaim (optimize sb-cover:store-coverage-data :quality 3))

(defun test (n)
  (when (zerop n)
    (if (eql n 0)
      (print 'zero)
      (if (eql n 0.0)
        (print 'single-fp-zero)
        (print 'double-fp-zero))))
  (when (minusp n)
    (print 'negative))
  (when (plusp n)
    (tagbody
      (print 'positive)
      (go end)
      (print 'dummy)
      end)))

(test 0)
(test 1)
```

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
(requiring :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.
(prove: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**

**Travis CI and Coveralls**

[Travis](https://travis-ci.org) is a service for running unit tests in the cloud and [Coveralls](https://coveralls.io) shows you the evolution of coverage over time, and also tells you what a pull request will do to coverage.

Thanks to cl-travis we can easily test our program against one or many Lisp implementations (ABCL, Allegro CL, SBCL, CMUCL, CCL and ECL). cl-coveralls helps to post our coverage to the service. It supports SBCL and Clozure CL with Travis CI and Circle CI.
We refer you to the lengthy and illustrated explanations of the “continuous integration” page on lisp-lang.org.

You’ll find many example projects using them in the links above, but if you want a quick overview of what it looks like:

- Lucerne on Coveralls and Travis.

**Gitlab CI**

**Gitlab CI** is part of Gitlab and is available on [Gitlab.com](https://gitlab.com), for public and private repositories. Let’s see straight away a simple `.gitlab-ci.yml`:

```yaml
image: daewok/lisp-devel

before_script:
  - apt-get update -qy
  - apt-get install -y git-core
  - 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 `daewok/lisp-devel` one. It includes SBCL, ECL, CCL and ABCL, and Quicklisp is installed in the home (`/home/lisp/`), so we can quickload packages right away. If you’re interested it also has a more bare bones option. 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 clone a library where Quicklisp can find it, and for doing so we must install git (Docker images are usually pretty bare bones).

We can try locally ourselves. If we already installed [Docker](https://www.docker.com) 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

docker run --rm -it -v /path/to/local/code:/usr/local/share/common-lisp:latest bash

This will download the lisp image (±400Mo), mount some local code in the image where indicated, and drop us in bash. Now we can try a make test.

To show you a more complete example:

image: daewok/lisp-devel

stages:
  - test
  - build

before_script:
  - apt-get update -qy
  - apt-get install -y git-core
  - git clone https://github.com/foo/bar ~/quicklisp/local-projects/

test:
  stage: test
  script:
    - make test

build:
  stage: build
  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

When the pipelines pass, you will see:
You now have a ready to use Gitlab CI.
Database Access and Persistence

The [Database section on the Awesome-cl list](https://github.com/awesome-cl/database) 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](https://github.com/awesome-cl/mapping) (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](https://github.com/awesome-cl/mapping) is “an ORM for Common Lisp with migrations, relationships and PostgreSQL support”.

- it supports [MySQL](https://dev.mysql.com), [PostgreSQL](https://www.postgresql.org) and [SQLite3](https://www.sqlite.org),
- 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 "fuka"
```

The driver type can be of :mysql, :sqlite3 and :postgres.

With sqlite you don’t need the username and password:

```
(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."
  (connect-toplevel :sqlite3 :database-name "myapp"))

Models

Defining models

In Mito, you can define a class which corresponds to a database table by specifying (:metaclass mito:dao-table-class):

(defclass user ()
  ((name :col-type (:varchar 64)
    :initarg :name
    :accessor user-name)
   (email :col-type (or (:varchar 128) :null)
     :initarg :email
     :accessor user-email))
 (:metaclass mito:dao-table-class))

Note that the class automatically adds some slots: a primary key named id if there’s no primary keys, created_at and updated_at for recording timestamps. To disable these behaviors, specify :auto-pk nil or :record-timestamps nil to the defclass forms.

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::NAME
    ;   #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS COMMON-LISP-USER::EMAIL
    ;   #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS MITO.DAO.MIXIN::CREATE
    ;   #<MITO.DAO.COLUMN:DAO-TABLE-COLUMN-CLASS MITO.DAO.MIXIN::UPDATE

The class inherits mito:dao-class implicitly.

(find-class 'user)
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

Field types are:

(:varchar <integer>),

:serial, :bigserial, :integer, :bigint, :unsigned,
Optional fields

Use (or <real type> :null):

(email :col-type (or (:varchar 128) :null)
 :initarg :email
 :accessor user-email))

Field constraints

:unique-keys can be used like so:

(defclass user ()
 ((name :col-type (:varchar 64)
   :initarg :name
   :accessor user-name)
 (email :col-type (:varchar 128)
   :initarg :email
   :accessor user-email))
 (:metaclass mito:dao-table-class)
 (: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:

(defclass tweet ()
 ((status :col-type :text
   :initarg :status
   :accessor tweet-status)
 ;; This slot refers to USER class
(user :col-type user
  :initarg :user
  :accessor tweet-user))
(:metaclass mito:dao-table-class))

(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:

```lisp
(defclass book ()
  ((title
     :col-type (:varchar 128)
     :initarg :title
     :accessor title)
   (ean
     :col-type (or (:varchar 128) :null)
     :initarg :ean
     :accessor ean))
 (:metaclass mito:dao-table-class))
```

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:

```lisp
(defclass user-books ()
  ((user
     :col-type user
     :initarg :user)
   (book
     :col-type book
     :initarg :book))
 (:metaclass mito:dao-table-class))
```

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:

```lisp
(defclass user-books ()
```

```lisp
```

```lisp
```
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:

```scheme
(defclass user ()
  ((name :col-type (:varchar 64)
    :initarg :name
    :accessor user-name)
   (email :col-type (:varchar 128)
    :initarg :email
    :accessor user-email))
  (:metaclass mito:dao-table-class)
  (:unique-keys email))

(defclass temporary-user (user)
  ((registered-at :col-type :timestamp
    :initarg :registered-at
    :accessor temporary-user-registered-at))
  (:metaclass mito:dao-table-class))

(mito:table-definition 'temporary-user)
;; => (S$QL-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 doesn’t related to any database tables, you can use DAO-TABLE-MIXIN. Below the has-email class will not create a table.

```lisp
(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>

(defclass user (has-email)
  ((name :col-type (:varchar 64)
    :initarg :name
    :accessor user-name))
  (:metaclass mito:dao-table-class))
=> #<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

First create the tables if needed:

(ensure-table-exists 'user)

then alter the tables, if needed:

(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)

There are no changes from the previous user definition:
Now let's add a unique email field:

```lisp
(defclass user ()
  ((name :col-type (:varchar 64)
    :initarg :name
    :accessor user-name)
   (email :col-type (:varchar 128)
    :initarg :email
    :accessor user-email))
  (:metaclass mito:dao-table-class)
  (:unique-keys email))
```

The migration will run the following code:

```lisp
(mito:migration-expressions 'user)
;=> (#<SXQL-STATEMENT: ALTER TABLE user ALTER COLUMN email TYPE character varying(128), ALTER COLUMN email SET NOT NULL>
    #<SXQL-STATEMENT: CREATE UNIQUE INDEX unique_user_email ON user>)
```

so let's apply it:

```lisp
(mito:migrate-table 'user)
;-> ;; ALTER TABLE "user" ALTER COLUMN "email" TYPE character varying
    ;; CREATE UNIQUE INDEX "unique_user_email" ON "user" ("email")
    ;; ALTER TABLE user ALTER COLUMN email TYPE char
    #<SXQL-STATEMENT: CREATE UNIQUE INDEX unique_user_email ON user>
```

**Queries**

**Creating objects**

We can create user objects with the regular `make-instance`:

```lisp
(defvar me
  (make-instance user :name "Jane Doe" :email "jane.doe@example.com")
  ...)
```
To save it in DB, use insert-dao:

```
(mito:insert-dao me)
;; INSERT INTO `user` (`name`, `email`, `created_at`, `updated_at`)
;=> #<USER {10053C4453}>
```

Do the two steps above at once:

```
(mito:create-dao 'user :name "Eitaro Fukamachi" :email "e.arrows@gma
```nYou 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:

```
(defvar 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.DAO:

;;;; or:
(mito:delete-by-values 'user :id 1)
;;;; DELETE FROM `user` WHERE (`id` = ?) (1) [0 rows] | MITO.DAO:

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] | M
;;;;=> #<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 of th
  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
Find all

Use the macro select-dao.

Get a list of all users:

```mito
(mito:select-dao 'user)
;(#{USER {10077C6073}})
;#{SXQL-STATEMENT: SELECT * FROM user>
```

Find by relationship

As seen above:

```mito
(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:

```mito
(select-dao 'tweet
  (where (:like :status "%Japan%"))
```

another:

```mito
(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:

```lisp
(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.

**Clauses**

See the SxQL documentation.

**Examples:**

```lisp
(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
:==, :!=
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@gma
 ;-> Adding "Eitaro Fukamachi"...
 ; ;; INSERT INTO "user" ("name", "email", "created_at", "updated_a
 ;=> #<USER {100835FB33}>

Inflation/Deflation

Inflation/Deflation is a function to convert values between Mito and RDBMS.

(defclass user-report ()
  ((title :col-type (:varchar 100)
    :initarg :title
    :accessor report-title)
   (body :col-type :text
     :initarg :body
     :initform ""
     :accessor report-body)
   (reported-at :col-type :timestamp
     :initarg :reported-at
     :initform (local-time:now)
     :accessor report-reported-at
     :inflate #'local-time:universal-to-timestamp)
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%") [3
  ;-> ;; SELECT * FROM `user` WHERE (`id` IN (?, ?, ?)) (1, 3, 12) [3
  ;-> (#<TWEET {1003513EC3}> #<TWEET {1007BABEF3}> #<TWEET {1007BB9D63

  ;; 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",
  -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'
  -D, --directory DIRECTORY  Directory path to keep migration
  --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))

(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 "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"))
    ;; ...
(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 new DB
      (let* ((*db-name* name))
        ;; Always re-connect to our real DB even in case of error i
        (unwind-protect
          (progn
            ;; our functions to connect to the DB, create the table
            (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))
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
- an advanced "defmodel" macro.
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
- Qt4 with Qtools
- IUP with lisnik/iup
- Gtk3 with cl-cffi-gtk
- 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 x86 Linux platforms, and on the Mac.
- **CCL’s built-in Cocoa interface**, used to build applications such as Opusmodus.
- **CocoaInterface**, a Cocoa interface for Clozure Common Lisp. Build Cocoa user interface windows dynamically using Lisp code and bypass the typical Xcode processes.
- **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.
- **nodgui**, a fork of Ltk, with syntax sugar and additional widgets.
- **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](http://www.shadereel.com/abcl/api/index.html)
- 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)**

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:

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.

So, Tk isn’t fancy, but it is an 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.

The Lisp binding is Ltk.

- **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 calendar. We can find some in extensions (such as in Nodgui), but they don’t feel native, at all.
- **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 to non-existent.
  - **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-torrents - searching torrents on popular trackers. CLI, readline and a simple Tk GUI.

**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 autocomplet 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 yet to be created.  

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:

**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.
- **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, ColorBro FlatSeparator, Link, List, FlatList, ProgressBar, Spin, Text, Toggle listDialog, Alarm, Color, Message, Font, Scintilla, file-dialog... Cells, Matrix, MatrixEx, MatrixList, GLCanvas, Plot, MglPlot, OleControl, WebBrowser (WebKit/Gtk+)... drag-and-drop
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, (collapsible) tree, list, grid, date-picker

![Widgets](image)

**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:

```lisp
(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`:

```lisp
(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:

```lisp
(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)
```

**Qt4**

```
(ql:quickload '(:qtools :qtc core :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-window) 
(setf (q+:placeholder-text name) "Your name please.")

(define-subwidget (main-window go-button) (q+:make-qpushbutton "Go!")

We stack them horizontally:

(define-subwidget (main-window layout) (q+:make-qhboxlayout main-window) 
(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!" (q+

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) ()
```
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 "G
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](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
```
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

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"))))
```

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
(iup:version)
(lisp-implementation-typ
(lisp-implementation-ver
(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:

(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).

```
(iup:button :title "Test &1"
  :expand :yes
  :tip "Callback inline at control creation"
  :action (lambda (handle)
    (iup:message "title" "button1's action callback"
      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
                                          (iup:version)
                                          (lisp-implementation-typ
                                          (lisp-implementation-ver
                                          (button (iup:button :title "Click me"
                                          :expand :yes
                                          :tip "yes, click me"
                                          :action (lambda (handle)
                                          (declare (ignoreable 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))))
```


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.

```lisp
(defun counter ()
  (iup:with-iup ()
    (let* ((counter (iup:label :title 0))
           (label (iup:label :title (format nil "The button was clicked ~a time(s)." iup:attribute counter))
           (button (iup:button :title "Click me"
                               :expand :yes
                               :tip "yes, click me"
                               :action (lambda (handle)
                                         (declare (ignore handle)
                                                  (setf (iup:attribute counte
                                                     (1+ (iup:attribute counter))
                                                  (setf (iup:attribute label
                                                     (format nil "The butt
                                                     (iup:attribute 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 sel
                             :tip "List 2" :expand :yes))
           (list-3 (iup:list :value 9 :tip "List 3" :expand :yes))
           (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 exam

    (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:

```

Uncomment and evaluate this line only if you want to enable the OpenGL 2 renderer:

```
(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.

```
(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))
```
(defun run ()
  (bodge-host:open-window (make-instance 'main-window)))

and run it:

(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, Iquery, …), 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" "caveman" "spinneret" "djula"))

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 4
(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 root/ 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>
```
Let’s start a new acceptor on a new port:

```
(defun *my-acceptor* (make-instance 'hunchentoot:easy-acceptor :port :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 4)
```

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-regexp-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 424
  (hunchentoot:start *server*)


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

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 <uri> ...) (lambda list parame

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))

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)`

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 HTTP method (otherwise cumbersome in Hunchentoot)
- arguments extraction from the url path
- and decorators.

To use it, don’t create a server with `hunchentoot:easy-acceptor` but with `easy-routes:easy-routes-acceptor`:

`(setq *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:

```lisp
(easy-routes:defroute name "/foo/:x" :method :get) (y &get z)
  (format nil "x: ~a y: ~a z: ~a" x y 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.

**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:

```lisp
(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:

```lisp
(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](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.
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 are *catch-errors-p*, *show-lisp-errors-p* and *show-lisp-backtraces-p*.

Hunchentoot also defines condition classes.

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 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.

Webblocks 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.

Webblock’s unit of work is the widget. They look like a class definition:

```
(defwidget task ()
  ((title
    :initarg :title
    :accessor title)

```
Then all we have to do is to define the `render` method for this widget:

```lisp
(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:

```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.


**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 "webapp"

```

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.html"

```

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:
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:

```lisp
(push (hunchentoot:create-folder-dispatcher-and-handler
       "/static/" (merge-pathnames "src/static" ;; starts without a
       (asdf:system-source-directory :my
       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:

```lisp
(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:

```lisp
(defun logged-in-p ()
  (gethash :user cm:*session*))
```

and we define our with-logged-in macro:

```lisp
(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:

(dfroute "/account/logout" ()
 "Show the log-out page, only if the user is logged in."
 (with-logged-in
  (logout)
  (render #p"logout.html"))
)

(dfroute ("/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$19c7f2cccb38"
(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.

```lisp
(defun password-hash (password)
  (ironclad:pbkdf2-hash-password-to-combined-string
   (babel:string-to-octets password)))
```

`pbkdf2` is defined in [RFC2898](https://tools.ietf.org/html/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.

```lisp
(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:string-to-octets password) 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.

```lisp
(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
                       :id_user)))))
    db))
```
Credit: /u/arvid on /r/learnlisp.

Running 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:

```lisp
;; 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 (error c)
    (format *error-output* "~&An error occurred: ~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)
```
(defvar window (ceramic:make-window :url "http://localhost:8000/"))

(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 ©.

**Daemonizing, restarting in case of crashes, handling logs with Systemd**

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:

```
$ emacs -nw /etc/systemd/system/my-app.service
[Unit]
Description=stupid simple example

[Service]
WorkingDirectory=/path/to/your/app
ExecStart=/usr/local/bin/sthg sthg
```
Then we have a command to start it:

```
sudo systemctl start my-app.service
```

a command to check its status:

```
systemctl status my-app.service
```

and Systemd can handle **logging** (we write to stdout or stderr, it writes logs):

```
journalctl -f -u my-app.service
```

and it handles crashes and **restarts the app**:

```
Restart=always
```

and it can **start the app after a reboot**:

```
[Install]
WantedBy=basic.target
```

to enable it:

```
sudo systemctl enable my-app.service
```

**With Docker**

There are several Docker images for Common Lisp. For example:

- **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 Next browser.

**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), \n  (ql:quickload :quickutil-server) (ql:quickload :swank-client
  (swank-client:with-slime-connection (conn "localhost" $(SWAN
  (swank-client:slime-eval (quote (handler-bind ((error (f
  (ql:quickload :quickutil-utilities) (ql:quickload :q
  (funcall (symbol-function (intern "STOP" :quickutil-
  (funcall (symbol-function (intern "START" :quickutil
  $($(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/](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:

```lisp
(ql:quickload '("dexador" "plump" "lquery" "lparallel"))
```

HTTP Requests

Easy things first. Install Dexador. Then we use the get function:

```lisp
(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>

```

"<!DOCTYPE html>
<html lang="en">
<head>

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/](https://shinmera.github.io/lquery/)

We first need to parse the html into an internal data structure. Use `(lquery:$ (initialize <html>))`:

```lisp
(defvar *parsed-content* (lquery:$ (initialize *request*)))
```

lquery uses 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:

```lisp
(lquery:$ *parsed-content* "#content li")
```

;; => #(<plump-dom:element li {100b3263a3}> #(<plump-dom:element li {100b326423}> #(<plump-dom:element li {100b3264a3}> #(<plump-dom:element li {100b3264d3}> #(<plump-dom:element li {100b326543}> #(<plump-dom:element li {100b3265b3}>
```
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 Table"
 "Pattern Matching / Regular Expressions" "Functions" "Loop" "Input"
 "Files and Directories" "Packages" "Macros and Backquote"
 "CLOS (the Common Lisp Object System)" "Sockets" "Interfacing with"
 "Foreign Function Interfaces" "Threads" "Defining Systems"
 [...]
 "Pascal Costanza’s Highly Opinionated Guide to Lisp"
 "Loving Lisp - the Savvy Programmer’s Secret Weapon by Mark Watson"
 "FranzInc, a company selling Common Lisp and Graph Database soluti

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" "dates_a
;; "hashes.html" "pattern_matching.html" "functions.html" "loop.htm
;; "files.html" "packages.html" "macros.html"
;; "/cl-cookbook/clos-tutorial/index.html" "os.html" "ffi.html"
;; "process.html" "systems.html" "win32.html" "testing.html" "misc.
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:

```lisp
(defvar *urls* (lquery:$ *parsed-content* "#content li a" (attr :hr)
```

We remove the elements that start with “mailto:”: (a quick look at the strings page will help)

```lisp
(remove-if (lambda (it) (string= it "mailto:" :start1 0 :end1 (length it))
```
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 it)))) *urls*
```

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") ;; too bad indeed
```

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.)*

```lisp
(map 'vector (lambda (it)
   (ignore-errors
      (nth-value 1 (dex:get it))))
   *filtered-urls*)
```

we get:

#(NIL 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200 200)

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 and 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:

```lisp
(time (lparallel:pmap 'vector
   (lambda (it)
      (ignore-errors (let ((status (nth-value 1 (dex:get it)))) status
       *filtered-urls*))
   )); Evaluation took:
```
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:

```lisp
... (if (and status (= 200 status)) it) ...
(defvar *valid-urls* *)
```

we get a vector of urls with a couple of nils: 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.

```lisp
(set-difference (coerce *filtered-urls* 'list)
(coerce *valid-urls* 'list))
```

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:

```lisp
(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:

```lisp
;;; 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.

```lisp
(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))))
You may now start your server, running on port 12345:

```lisp
(defun start-once (ws)
  (websocket-driver:on :open ws
    (lambda (&key code reason)
      (declare (ignore code reason))
      (handle-open-connection ws)))

  (lambda (responder)
    (declare (ignore responder))
    (websocket-driver:start-connection ws)))
```

---

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:

```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").appendChild(li);
}

const ws = new WebSocket("ws://localhost:12345/chat");
wsi.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 8080))

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>
  </body>
  
  <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").appendChild(li);
      }

      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 = "";
        }
      });
    });
(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 8080))
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
- Fernando Borretti
- bill_clementson
- chuchana
- Ben Dudson
- Pierre Neidhardt
- Rommel MARTINEZ
- digikar99
- nicklevine
- Danny YUE
- Dmitry Petrov
- otjura
- YUE Daian
- skeptomai
- thegoofist
- Francis St-Amour
- emres
- jdcal
- Boutade
- airfoyle
- mvilleneuve
- Alex Ponomarev
- Alexander Artemenko
- Johan Sjölén
- Mariano Montone
- albertoriva
- Nisar Ahmad
- Nisen
- Vityok
- ozten
- Amol Dosanjh
- Andrew
- Andrew Hill
- Bibek Panthi
- Bo Yao
- Burhanuddin Baharuddin
- Coin Okay
- Danny Yue
- HiPhish
- Kevin Layer
- LdBeth
- Matthew Kennedy
- Momozor
- Paul Donnelly
- Pavel Kulyov
- R Primus
- Salad Tea
- Victor Anyakin
- convert-repo
- jthing
- mwkgkg
- 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
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 ;)

---

- 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**