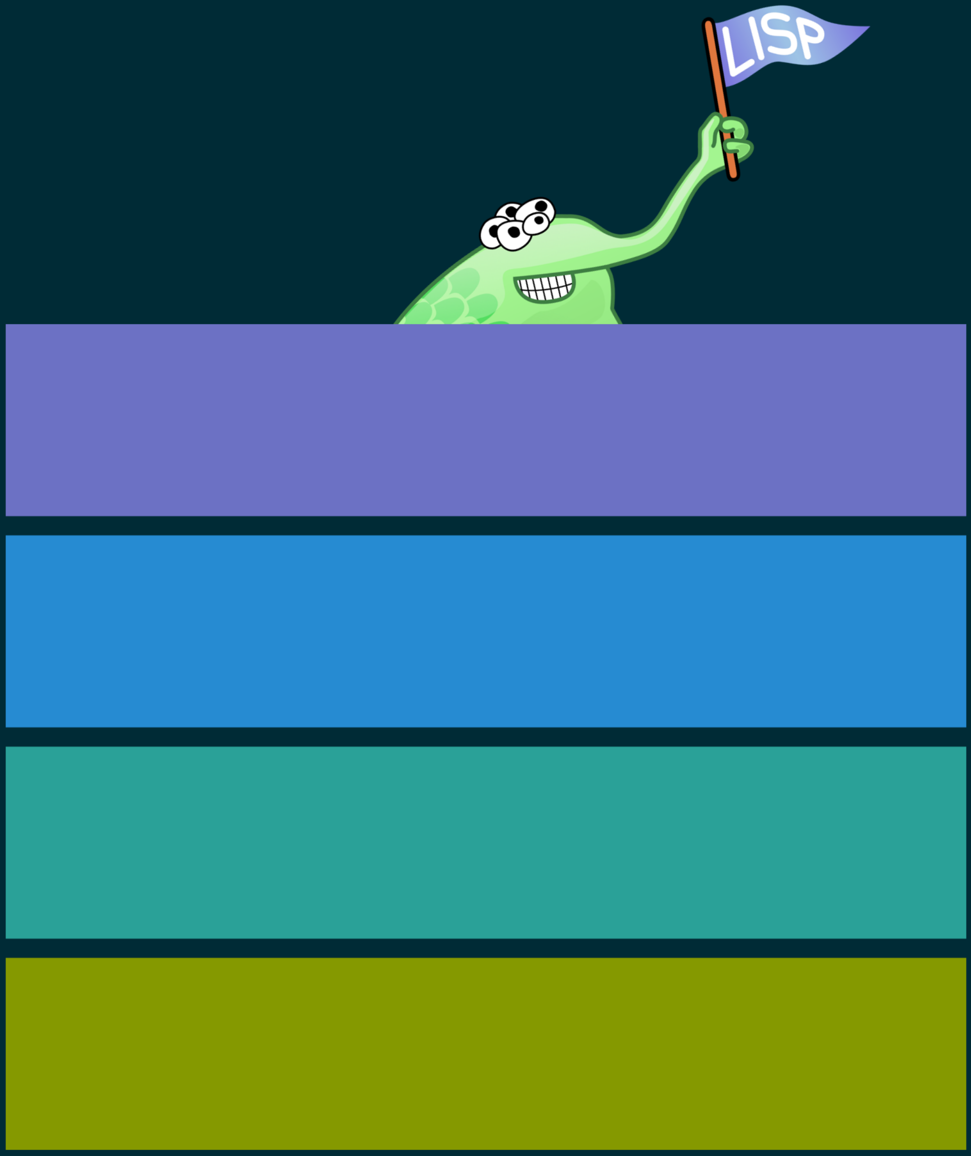
Full Stack Lisp

*Build and deploy modern Lisp applications **



By Pavel Penev

* Buzzword Compliant

Full Stack Lisp

Build and deploy modern Lisp applications

Pavel Penev

This book is for sale at http://leanpub.com/fullstacklisp

This version was published on 2016-05-30

Tweet This Book!

Please help Pavel Penev by spreading the word about this book on Twitter!

The suggested hashtag for this book is #FullStackLisp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#FullStackLisp

Contents

Warning to early readers . . . . . **i**
Current Status i
What remains to be written ii
Additional ideas ii
Thank you! ii

About . . . . . **iii**

Introduction . . . . . **iv**

Prerequisites . . . . . **v**

Typographic conventions . . . . . **vi**

Part One: Development environment **1**

Chapter 1: Roswell **2**
A quick word about Lisp implementations 2
Installing Roswell 3
Installing Lisp implementations 3
A quick primer on Scripting with Roswell 5

Chapter 2: Lisp libraries **9**
ASDF: Another System Definition Facility 9
Quicklisp 10
Local projects 11
Project generators 11
Commonly used libraries 11

Chapter 3: Editing Lisp **14**
Emacs 14
Slime 19
Sly 26

Warning to early readers

This is a very early work in progress version of Full Stack Lisp. It is incomplete and possibly contains incorrect information. I'm releasing it as a status update and to get early feedback. Constructive criticism is extremely welcome. I hang out in irc on freenode in `#lisp` and `#lispweb` and on twitter @pavelpenev[1]. The book also has a dedicated twitter account @fullstacklisp[2] and a web page[3]. Source code is on github[4]. If you find anything wrong with it please open an issue.

Current Status

Each section of the book is in one of the following states(ordered by proximity to completion):

- To be written
- Early draft
- Late draft
- Needs editing
- Done

The current status of book is as follows

- Frontmatter (Needs editing)
- Part I
  - Chapter 1: Roswell (Late Draft)
  - Chapter 2: Lisp libraries (Early draft)
    Some parts are missing and will possibly be expanded beyond that at a later stage
  - Chapter 3: Editing Lisp (Early draft)
    Many sections are TBW, some parts might get rewritten. Additionally I'll eventually cover the Atom editor.
- Part II
  - Chapter 4: Caveman2 (Early draft)
    Very early draft, possibly subject to a complete rewrite. Many missing parts.

[1] https://twitter.com/pavelpenev
[2] https://twitter.com/fullstacklisp
[3] http://fullstacklisp.com
[4] https://github.com/pvlpenev/fullstacklisp

What remains to be written

Current plans include:

- A chapter walking the reader through developing a simple wiki app

  Will include detailed information on Caveman2's features, including a detailed Djula tutorial, more Envy examples, and an introduction to it's DB layer.
- A chapter on alternatives

  Caveman uses Djula for templating and datafly/sxql for DB access. We'll introduce alternatives, such as s-expression based HTML generation and various database libraries
- A basic Ironclad tutorial

  Introduction to ironclad and how to properly store passwords, API keys, ect.
- A chapter on frontend tools.

  Tools to help with CSS and writing JavaScript such as paredit
- A chapter on deployment

  From a simple Nginx setup to using docker with common lisp

Additional ideas

I have some other ideas that might not make it into the book, such as:

- A chapter on Restful services
- Building desktop apps in a webview
- Packaging executables

If you have other ideas and they seem like they might be in the scope of the book, I'd love to hear from you!

Thank you!

Thank you for reading this early draft! Feedback is extremely crucial at this phase of the project. I might be painting myself into a corner and the only way to know is if somebody tells me. Thank you especially if you bought the book through leanpub. I'm currently self-funding this project, which is a fancy way of saying I'm burning through my savings to write and I don't know how much I'll be able to finish before they run out. Your contribution is greatly appreciated and is of great help!

About

Lisp aliens have infected the terrestrial memetosphere with the meme complex of Common Lisp. The mindless human hosts find themselves helpless spreaders of the glory and joy of Lisp Development. Join us and become a great Lisp Programmer! Learn how to build and deploy modern Lisp applications in the booming ecosystem of modern software inhabited by lesser technologies. In the name of alien memetic symbiosis, Go Forth And Hack!

Introduction

After a couple of years of playing around with Scheme and later Clojure around 2008-2010 I made the fateful decision to learn Common Lisp. Originally the plan was to learn just enough Lisp to be able to read "On Lisp" by Paul Graham with proficiency and increase my mad Clojure skills, which was my favorite language at the time. My first reaction to this mutable and downright archaic dialect was a mix between fascination and revulsion. Uppercase symbols, uuuugghhhh! Somehow this mess of a language grew on me and I've been in love ever since. I've half joked about it infecting my brain, and if you think of a programming language as a memetic complex, it obviously is capable of infecting human hosts, so I'm not sure the joke isn't an actual fact. Send Help!

When I look back on the last five years I've been involved in this horrible mess, the amount of change is astonishing. Lisp might be the mother of all late bloomers, well into its fifties it grew into a viable platform for actual applications written by mortal programmers. The Common Lisp ecosystem is growing into a very usable state, it's a great time to explore how to actually use this language in a practical manner from top to bottom and deliver modern applications.

This is a book about application development. That means we'll explore run-times, servers, storage, frameworks and deployment, as well as various ways to use Lisp as part of a client application. In addition tools, libraries and practices the author finds useful will find their way into the book.

This book assumes only basic Lisp knowledge, complete newbies should read an introductory tutorial before reading this book.

The following is a quick outline of some of the topics this book will cover:

- Setting up a development environment
- Web servers, Clack HTTP library and the various frameworks built on top of it.
- Client side lisp ** Lispy HTML/CSS/JavaScript ** Using 3rd party APIs ** Desktop apps
- DBs and ORMs
- Very basic security and crypto
- Deployment and management

Prerequisites

I’m asuming you’re at least somewhat familiar with lisp. If not, you should read a tutorial or an introductory book such as “Practical Common Lisp” or “Land of Lisp”. This book isn’t meant for complete beginners, but you don’t need to know much to get started and pick things as you go along.

You should also have a basic understanding of web development. Terms like HTTP, HTML, web servers, etc. shouldn’t be foreign words to you.

You should have a familiarity with Unix-like operating systems, like Linux, OS X or the BSDs.

Some database knowledge will be extremely useful in later chapters.

As always, your google-fu is your greatest friend.

Typographic conventions

These are the typographic conventions used in this book.

Code appearing inline in text paragraphs looks like this:

This code is inlined: `(lambda () (format t "Hello World"))`.

This is a code block in a file.:

```
(defun hello-world ()
  (format t "Hello World"))
```

The following characters represent various prompts:

A `*` represents a Lisp REPL(Read Eval Print Loop, the lisp interactive interpreter), `=>` marks the returned result:

```
 * (format nil "Hello World")
 => "Hello World"
```

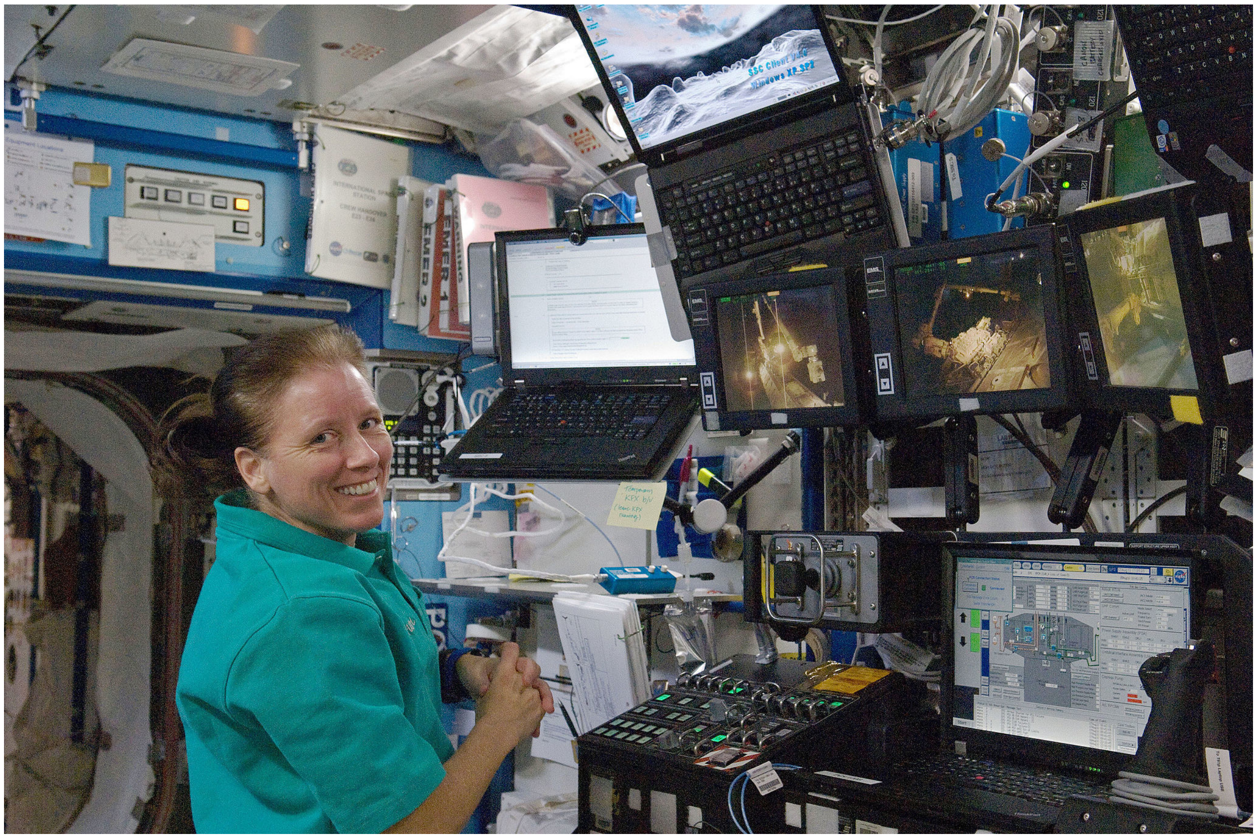
`$` is a unix shell, `#` is a root shell, or code executed with `sudo`:

```
 # apt-get install foo
 $ foo --bar baz
```

`>` is a windows `cmd.exe` prompt:

```
 > dir C:\
```

Part One: Development environment



NASA astronaut Shannon Walker, Expedition 24 flight engineer, is pictured near a robotic workstation in the Destiny laboratory of the International Space Station. One day I'll have a workstation as cool as hers. LICENCE: Public Domain.

Source[5]

In this part of the book we'll talk about everything you need in order to set up and use a basic Lisp and Emacs environment. We'll cover installing Lisp, writing scripts, editing lisp and interacting with the REPL as well as an introduction to some utilities and commonly used libraries that you should know about.

[5]https://commons.wikimedia.org/wiki/File:ISS-24_Shannon_Walker_in_front_of_the_robotic_workstation.jpg

Chapter 1: Roswell

Roswell is a command line interface to common lisp. It's a way for you to manage your lisp setup from the unix shell. For example it can be used to install lisp implementations, write and run scripts written in Lisp and produce executables among other things. It's fairly new, but it's unquestionably one of the most useful tools to come out recently.

A quick word about Lisp implementations

Because Lisp is an ANSI standard anybody is free to implement it themselves, and over the decades a significant number of implementations have appeared. In this section I'll list some of the more important implementations and why you might choose it over the others.

The most popular open source implementation is SBCL, it has a native compiler and generates very efficient code. With proper optimizations it is not unheard of to get C-like performance out of Lisp programs. SBCL happens to be the authors personal favorite.

CCL is probably the second most popular implementation. Like SBCL it generates native code, although it has a faster compiler(meaning compiling code takes less time). For this reason it might be a better choice for use during development, while SBCL might be better suited for production, although both work well in both cases. An additional feature of CCL is that on OS X it comes with it's own IDE and a cocoa bridge for GUI programming. Both are useful and very high-quality so in this book we'll show you how to set them both up.

Some honorable mentions:

ABCL is Common Lisp running on the JVM. If you need to interface to a java system or are familiar with that ecosystem, ABCL deserves a serious look.

ECL or Embeddable Common Lisp is a very small implementation which as the name suggests supports embedding in a C/C++ application. In addition to it's bytecode compiler it supports native compilation through an intermediate translation to C. It has a much smaller footprint that the other implementations and is generally very cool.

There are also the two major commercial implementations: LispWorks and Allegro Common Lisp. Both with native compilers and their own IDEs, GUI frameworks, and many other libraries. Both are rather expensive, but have limited free versions for trial purpose and might be worth checking out.

In summary, unless you're doing something specific or just like to play around with interesting systems, SBCL and/or CCL are the implementations you should care about for the moment.

Installing Roswell

The best way to install Roswell is from source:

First let's install its dependencies:

```
# apt-get install git build-essentials automake libcurl4-openssl-dev
```

Next we clone the Roswell repo and build it:

```
$ git clone -b release https://github.com/roswell/roswell.git
$ cd roswell
$ sh bootstrap
$ ./configure
$ make
# make install
```

Let's verify that we've installed it:

```
$ ros version
roswell 0.0.3.56(e0d9c43)
build with gcc (Ubuntu 4.9.2-10ubuntu13) 4.9.2
libcurl 7.38.0
lispdir='/usr/local/share/common-lisp/source/roswell/'
configdir='/home/pav/.roswell/'
```

Type `ros help` to get a complete list of commands and options.

Installing Lisp implementations

Let's install SBCL and CCL. This is done with the `ros install <implementation>/[<version>]` command. You can have multiple versions of the same implementation installed at the same time. At any one time roswell will use one as the default. Let's see what's available for installation:

```
$ ros help install
Usage: ros install impl [OPTIONS]

For more details on impl specific options, type:
 ros help install impl

Candidates impls for installation are:
ccl-bin
sbcl
sbcl-bin
clisp
ecl
```

Ok, we have two candidates for SBCL: `sbcl` and `sbcl-bin`. The difference is that `sbcl-bin` downloads and installs a pre-build binary, while installing `sbcl` will download the source and build it. You should preferably install sbcl from source since on some platforms the pre-build binary was build without threading support and we threads:

```
$ ros install sbcl
```

When it's done we can try to run a repl to see if the installation was successful:

```
$ ros run
* (+ 1 2)

3
```

It appears it is. Now let's install CCL:

```
$ ros install ccl-bin
```

If we have more than one implementation installed we can choose which one we want to use by suplying the `-L` option:

```
$ ros -L ccl-bin run
Welcome to Clozure Common Lisp Version 1.11-r16635  (LinuxX8664)!

CCL is developed and maintained by Clozure Associates. For more information
about CCL visit http://ccl.clozure.com.  To enquire about Clozure's Common Lisp
consulting services e-mail info@clozure.com or visit http://www.clozure.com.

? (+ 1 2)
3
```

It appears to work. If you want to change the default installation you can use the `use` command. First let's list the installed implementations and then change the default:

```
$ ros list installed
Installed implementations:
ccl-bin
ccl-bin/1.11
sbcl
sbcl/1.3.2
$ ros use ccl-bin
```

Now when we type `ros run` we'll get a CCL repl by default.

A quick primer on Scripting with Roswell

Roswell installable scripts

Roswell has the ability to install scripts other people have written and shared through Quicklisp(the Lisp package manager, we'll take a detailed look at it in the next chapter).

By default roswell stores its scripts in ~/`.roswell/bin`, let's add it to the path:

```
$ echo "export PATH=\"\$HOME/.roswell/bin:\$PATH\"" >> ~/.bashrc
```

Installing new scripts is done with the `install` command:

```
$ ros install <script name>
```

For a list of currently available scripts see here[6]

Writing your own scripts

We'll write a very simple example of an echo script that prints everything you give it back at you. Here's the code:

[6]https://github.com/roswell/roswell/wiki/2.1-List-of-Roswell-Installable-Scripts

~/.roswell/bin/ros-echo.ros

```
#!/bin/sh
#|-*- mode:lisp -*-|#
#|
exec ros -Q -- $0 "$@"
|#

#|
An echo script
|#

(defun main (&rest args)
  (format t "~{~A ~}~%" args))
```

Now we must make the script executable:

```
$ chmod +x ~/.roswell/bin/ros-echo.ros
```

Let's try it out:

```
$ ros-echo.ros hello
hello
```

How it works

The first five lines basically say how the script is to be executed. First it is run as a normal shell script, but then the script is executed with the ros command with the name of the script as a parameter and the rest of the arguments passed to it.

~/.roswell/bin/ros-echo.ros

```
#!/bin/sh
#|-*- mode:lisp -*-|#
#|
exec ros -Q -- $0 "$@"
|#
```

Roswell loads the code and executes the main function. In this case it must take arbitrary number of arguments so we define it with an argument list of `(&rest args)`. The format directive simply prints the each element of the args list with spaces between them and a blank line at the end. It's literally one of the simplest possible scripts.

~/.roswell/bin/ros-echo.ros

```lisp
(defun main (&rest args)
  (format t "~{~A ~}~%" args))
```

Compiling scripts

What about performance though? Lets see how fast our script is:

```
$ time ros-echo.ros hello
hello
ros-echo.ros hello 0,36s user 0,01s system 98% cpu 0,374 total
```

That isn't very good. Fortunately Roswell has a way build our script into a binary executable:

```
$ ros build ~/.roswell/bin/ros-echo.ros
```

This will create a binary executable with the same name, but without the .ros extension. Let's see how well it performs:

```
$ time ros-echo hello
hello
ros-echo hello 0,00s user 0,01s system 70% cpu 0,011 total
```

That's a sizable improvement.

Dealing with arguments

Let's modify the script to give it commands. Say we want to extend the script to capitalize, upcase or downcase it's input. It will look something like this:

```
$ ros-echo.ros hello world
hello world

$ ros-echo.ros --capitalize hello world
Hello World

$ ros-echo.ros --upcase hello world
HELLO WORLD

$ ros-echo.ros --downcase HELLO WORLD
hello world
```

Here's how the script would look like now:

~/.roswell/bin/ros-echo.ros

```
#!/bin/sh
#|-*- mode:lisp -*-|#
#|
exec ros -Q -- $0 "$@"
|#

#|
An echo script
|#

(defun main (&optional $1 &rest args)
  (let ((args (cond ((string-equal $1 "--capitalize") (mapcar #'string-capitaliz\
e args))
                    ((string-equal $1 "--upcase") (mapcar #'string-upcase args))
                    ((string-equal $1 "--downcase") (mapcar #'string-downcase ar\
gs))
                    (t (cons $1 args)))))
    (format t "~{~A ~}~%" args)))
```

First, note the arguments to the `main` function. The first argument to the script gets bound to `$1` and the rest to `args`. In the `cond` form, if `$1` is one of the valid commands, we transform `args` appropriately, otherwise we simply stick `$1` at the head of the list, since in that case it isn't a command and we want to echo it as well. This is a very simple example, but it gives an idea of what you could do.

If you want to see more fancy ways to write scripts, check out the source of the Roswell installable scripts here[7]. For more information about Roswell check out it's wiki[8].

[7]https://github.com/roswell/roswell/wiki/2.1-List-of-Roswell-Installable-Scripts
[8]https://github.com/roswell/roswell/wiki/

Chapter 2: Lisp libraries

In this chapter we'll introduce the ASDF build system, Quicklisp the Lisp package manager as well as several commonly used libraries you should know about.

ASDF: Another System Definition Facility

Lisp code is organized in something called a system. A system is a directory containing all of your source code and a system definition file(with the `.asd` extension) which tells ASDF how to build and load your project. Almost all modern lisp projects use it.

Let's look at a very minimal ASDF project to get an idea of what we're talking about:

```
example-project/
  README.md
  example-project.asd
  src/
    example-project.lisp
```

In the `example-project` directory there are several files, the `README.md` is self-explanatory. The `src` directory contains all of our source files and the example-project.asd tells lisp how to build it. Let's take a look inside

example-project.asd

```
(in-package :cl-user)
(defpackage example-project-asd
  (:use :cl :asdf))
(in-package :example-project-asd)

(defsystem example-project
  :version "0.1"
  :author "Pavel Penev"
  :mailto "example@test.com"
  :license "MIT"
  :depends-on (:iterate :alexandria)
  :components ((:module "src"
                :components
                ((:file "example-project"))))
  :description "Example project")
```

As you can see this is just ordinary lisp code. Let's look at the `defsystem` form in more detail, it defines information about the project such as it's version, author, license, etc. The interesting parts are `:depends-on` and `:components`. The `:depends-on` clause lists the libraries our project depends on, obviously, and the components tells where to find the source code, in this case the file `example-project`(note it is written without the `.lisp` extension) in the `src` directory. More complicated set-ups exist and we'll look at them in time. For now this will suffice. In the next section we'll take a look at Quicklisp and how we can load such systems into Lisp.

Quicklisp

Quicklisp is a package manager for lisp. It handles downloading and installation of lisp code. If you're using roswell it's already installed on your system. Let's try it out, Open a lisp repl with `ros run` and type the following to install the alexandria(a collection of useful utilities) library:

```
 * (ql:quickload "alexandria")
 => ("alexandria")
```

Alexandria includes a function which takes a nested list and "flattens it" into a non-nested one, let's see if we installed alexandria by trying it out

```
 * (alexandria:flatten '((1 2) (3 4) (5 6)))
 => (1 2 3 4 5 6)
```

Here are some other useful Quicklisp commands:

`ql:system-apropos` takes a string and searches the names of available systems with it, for example if we search for "json" we get:

```
 * (ql:system-apropos "json")
=> #<SYSTEM cl-json / cl-json-20141217-git / quicklisp 2016-02-08>
   #<SYSTEM cl-json.test / cl-json-20141217-git / quicklisp 2016-02-08>
   #<SYSTEM com.gigamonkeys.json / monkeylib-json-20120208-git / quicklisp 2016-\
02-08>
   #<SYSTEM define-json-expander / define-json-expander-20140713-git / quicklisp\
 2016-02-08>
   #<SYSTEM graph-json / graph-20150407-git / quicklisp 2016-02-08>
   #<SYSTEM json-responses / json-responses-20151031-hg / quicklisp 2016-02-08>
   #<SYSTEM json-responses-test / json-responses-20151031-hg / quicklisp 2016-02\
-08>
   #<SYSTEM json-streams / json-streams-20140713-git / quicklisp 2016-02-08>
   #<SYSTEM json-streams-tests / json-streams-20140713-git / quicklisp 2016-02-0\
 8>
   #<SYSTEM json-template / cl-json-template-20110418-hg / quicklisp 2016-02-08>
   #<SYSTEM st-json / st-json-20150608-git / quicklisp 2016-02-08>
```

All of these systems can be loaded with `ql:quickload`.

The function `ql:update-client` will update your installation of Quicklisp.

Quicklisp libraries are usually updated once a month, occasionally you'll have to run `(ql:update-all-dists)` to get the newest versions.

The function `ql:uninstall` can be used to remove an installed system.

That about covers the basic Quicklisp usage.

Local projects

Quicklisp isn't only useful for installing remote libraries. It can also be used to load local ASDF systems. By default when you use `ql:quickload` Quicklisp searches the directories defined in the variable `ql:*local-project-directories*` for systems with the name you want to install, if they are not found there it tries to download them off the internet. If you use roswell among these directories will be `~/.roswell/local-projects`. This is where you can put your lisp code if you want it to be `quickload`-able.

TBW about `asdf:*central-registry*`

Project generators

TBW

Commonly used libraries

Below are a number of commonly used libraries and tools for Common Lisp. Some of them you might find useful, but mostly you should familiarize yourself with them, since you are likely to have to read code that uses them.

Quickdocs

Quickdocs[9] is a website that contains documentation for a vast majority of active Lisp projects. You can browse it to find interesting libraries.

[9]http://quickdocs.org/

Alexandria

Website[10] API[11]

Alexandria is possibly the most popular Lisp library. It's conservative collection of utilities meant to be portable across implementations. Quickdocs reports that 289 projects in Quicklisp use it. You will have to read code that uses it fairly often and it's way too useful not to know. I highly recommend reading it's documentation and becoming familiar with it. Here's some of the stuff in it:

- Hash table utilities
- Data and Control Flow
- Utilities to work with cons cell based data structures
- Sequence utilities
- Utilities to read files into strings or vectors
- Macro writing utilities
- Utilities for dealing with Symbols, Arrays, Types and Numbers

SPLIT-SEQUENCE

API[12]

The split-sequence library is very simple. It contains only 3 functions: split-sequence, split-sequence-if and split-sequence-if-not. Like alexandria, it's very commonly used.

Iterate

Website[13] API[14]

iterate is an iteration construct similar to the built in loop, but arguably better. It's very popular and even if you don't use it yourself you'll find yourself reading code that uses it very often, so it would be valuable if you get at least a basic understanding of its syntax.

Bordeaux-threads

Website[15] API[16]

Bordeaux-threads is a portable threading library for CL. Although the standard doesn't say anything about threads, most Lisp implementations provide them, unfortunately all hove minor differences. BT takes care of the problem by providing a common interface. If you do anything with threads, this library is a must.

[10]https://common-lisp.net/project/alexandria/
[11]https://common-lisp.net/project/alexandria/draft/alexandria.html
[12]http://quickdocs.org/split-sequence/
[13]https://common-lisp.net/project/iterate/
[14]https://common-lisp.net/project/iterate/doc/index.html
[15]https://common-lisp.net/project/bordeaux-threads/
[16]https://trac.common-lisp.net/bordeaux-threads/wiki/ApiDocumentation

Other useful library lists

cliki.net has a list of currently recommended libraries[17]

There is also Fernando Borretti's opinionated CL State of the Union[18] article which lists some recommendations

Finally you can look at Quicklisp download statistics to get an idea of what are some other commonly used libraries. The latest is from 2015-08-23[19]

[17]http://cliki.net/Current%20recommended%20libraries

[18]http://eudoxia.me/article/common-lisp-sotu-2015/

[19]http://blog.quicklisp.org/2015/08/july-2015-download-stats.html

Chapter 3: Editing Lisp

Because Lisp is a very interactive language in order to make full use of it, you need an interactive IDE. Users of the commercial Lisps(acl and LispWorks) can enjoy full feature IDEs, but in the Open Source world we are currently limited. The best options are Emacs and Vim with Slime and slimv(Superior Lisp interaction mode for Emacs/Superior Lisp interaction mode for Vim) or the new Emacs mode Sly(a fork of Slime with many improvements). A port of Slime is in progress for Atom, but it isn't very feature complete yet and it's github page claims it's still in Beta. If you don't want to use Emacs or Vim you might find this project interesting. Other than that there is the CCL IDE, but it only works on OS X. The last and least effective option is to write it like you would any other popular language. Write code in any editor you chose, compile manually, run and edit again. This option essentially nullifies a good chunk of Lisps advantages, at that point you might as well be writing in Blub. If you go with this option you will still have the power of the Common Lisp language, but not of the Common Lisp system. Lisp code wants to be developed interactively. Never the less, if you are a beginner and already use an editor that doesn't have an interactive lisp mode and you don't want to switch just yet, you can choose this option, at least for now. Eventually though you'll want to use the best tools available.

I highly recommend Emacs though. In this chapter I'll introduce it and Slime, as well as spend a bit of time discussing Sly as an alternative option. If Emacs isn't your thing you could skip this chapter.

Emacs

More than a mere text editor, Emacs is a rabbit hole of power. On the pyramid of editors, Emacs stands on top as the One True Editor. And the reason for that is very simple, having been born in the Lisp world it embodies the idea that a system should be completely modifiable, extensible and inspectable. Pretty much all of Emacs can be modified while it's running, it's self-documenting, you can pull up the documentation for every key binding, every interactive command and you can jump to the source of pretty much every part of the system and edit it(at least the parts written on Lisp, the core of Emacs is in C). If there was ever a tool designed to adapt to you, rather than you to it, it's Emacs. Throughout the decades millions of lines of Emacs Lisp code has been written in Emacs extensions. Everything from simple syntax high-lighting modes for programming languages to a full-blown text based web browser, email client, IRC client and who knows what else. People sometimes joke that Emacs is an OS and not an editor, but that joke is at least partially true. Other than common lisp itself, Emacs is one of the few bits of 80s Lisp Machines tech that survived into the modern world.

Getting started

TBW: Installation on various platforms

A quick note about notation: `Ctrl` and `Alt` is usually written as `C` and `M` in most Emacs tutorials, so `Ctrl-s` would be written as `C-s` and `Alt-x` would be written as `M-x`. The reason `M` is used is because on old systems `Alt` used to be called `Meta` and all emacs literature still calls it that. When you read the word `Meta` anywhere in relation to Emacs it simply means `Alt`. Sometimes to trigger a command more than one combination is used, for example saving a file is done with the key chord `C-x C-s`, which would mean you press the `Ctrl` key at the same time as `x`, and then you press `Ctrl` at the same time as `s`, alternatively, you can hold down `Ctrl` and then press `x` followed by `s`. This is a different command than `C-x s` in which case you type `C-x` followed by `s`. Emacs includes an interactive tutorial that will get you started with the most common editing commands as well as most of the relevant terminology. This is a prerequisite for the rest of the chapter. Once you start Emacs you can access it by typing `C-h t`(you press `C-h`, let go and then you type `t`). Spend some time with it and come back to this book. In the next section I'll show you how to customize Emacs.

A few more notes

As you've noticed in the tutorial the way you move around a file in Emacs is with the `C-p`, `C-n`, `C-b` and `C-f` keys. These do the same things as the arrow keys. It's a trade-off, unlike the arrow keys they are key combinations rather than a single stroke, but unlike the arrow keys they are on the keyboard itself rather than on the side, in other words you don't have to remove your hand from the typing position. Touch-typists will find the emacs key-bindings faster, especially with a little practice. Some users even disable the arrow keys. What I did wasn't as dramatic. I simply put a sticky note above my arrow keys and practiced using the emacs keys.

Another point to make is that the key-chords are in fact a bit awkward. Pressing `Ctrl` all the time could hurt your pinkie finger. There are two solutions to this problem. One is to rebind `Caps Lock` to `Ctrl`. Many emacs users do this since `Caps Lock` is closer to the home row of the keyboard and it makes tying key chords much easier and doesn't stress the pinkie as much. The other option is to swap the `Alt` and `Ctrl` keys. On the keyboards on which emacs was developed the `Ctrl` keys were on either side of the `Space` key and the user could press them with their thumbs rather than use their pinkies. This is the better option in my opinion. My own personal set up binds the `Alt` keys to `Ctrl` and I use the `Win` keys(with the Windows logo) to `Alt`. This is done in the KDE keyboard settings(the desktop environment I use).

TBW: Write an explanation on how to do these things for various platforms

A basic Emacs setup

The entry point for most Emacs customization is the file ~/`.emacs`. I've created an example `.emacs` file for your use based on my own customizations. You can find the file here[20]. I've included it below. It's heavily commented and I urge you to read it carefully before you decide to use it. It's part of this tutorial:

[20] https://gist.github.com/pvlpenev/079a4ad74111a99bb9ac

~/.emacs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; .emacs                                                                    ;;
;; Author: Pavel Penev                                                       ;;
;; Licence: MIT                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; This file describes a basic Emacs setup. Feel free to modify it to
;; your liking.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section I: Generic settings                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The following 3 lines disable unnecessary GUI elements, in this case the
;; menu bar, the tool bar and the scroll bar. If you wish, you can comment out
;; the menu-bar and keep it, but eventually I recommend you disable it.

(if (fboundp 'menu-bar-mode) (menu-bar-mode -1))
(if (fboundp 'tool-bar-mode) (tool-bar-mode -1))
(if (fboundp 'scroll-bar-mode) (scroll-bar-mode -1))

;; Sometimes the mini-buffer becomes multi-line, and it can be a bit annoying as
;; you type in it. This makes it stay one line.

(setq resize-mini-windows nil)

;; We don't need the Emacs splash screen. You can keep it on if you're into
;; that sort of thing

(setq inhibit-splash-screen t)

;;; global-linum-mode adds line numbers to all open windows, In my opinion
;;; there is no reason not to have this enabled.

(global-linum-mode)

;;; Emacs comes with a built-in text based browser. Since I use the browse-url
;;; function most often to browse documentation, I've set it to eww, the Emacs
;;; Web Browser. It works well for that purpose. If you would prefer to use a
;;; graphical browser, you can change this line.

```

```
(setq browse-url-browser-function 'eww-browse-url)

;;; I prefer to make the default font slightly smaller.

(set-face-attribute 'default nil :height 90)

;; Show matching parentecies globaly.

(show-paren-mode 1)

;; Use UTF-8 by default
(set-language-environment "UTF-8")

;; Don't ring the bell. It saves some annoyance

(setq ring-bell-function 'ignore)

;; If you're one of the heathens who prefers tabs over spaces, you should
;; remove the following line. It makes indentation use spaces.
(setq-default indent-tabs-mode nil)

;;; A simple backup setup. Makes sure I don't foo~ and #.foo files in
;;; directories with files you edit.

(setq
 backup-by-copying t      ; don't clobber symlinks-
 backup-directory-alist
 '(("." . "~/.saves"))    ; don't litter my fs tree
 auto-save-file-name-transforms
 `((".*" ,temporary-file-directory t))
 delete-old-versions t
 kept-new-versions 6
 kept-old-versions 2
 version-control t)       ; use versioned backups


;; Set up emacs server. This allows you to run emacs in the background and
;; connect to it with emacs client. It reduces startup time significantly. If
;; the server is not running, it starts it.

(load "server")

```

```
(unless (server-running-p)
  (server-start))

;;; ido-mode, or Interactively DO mode, adds lots of improvements when working
;;; with buffers of files. You can read more about it at:
;;; https://www.emacswiki.org/emacs-test/InteractivelyDoThings
(require 'ido)
(ido-mode t)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section II: Packages                                                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Elpa, the default package repository for emacs is fairly conservative, so
;; we'll add the melpa and marbaraie repositories
(require 'package)
(add-to-list 'package-archives
             '("marmalade" . "http://marmalade-repo.org/packages/"))

(add-to-list 'package-archives
             '("melpa" . "http://melpa.org/packages/") t)

;; You can install packages by typing M-x package-install <package-name>. I
;; recomend you install the following packages: smex, which adds an improved
;; version of M-x. I highly recomend this. You can read more about smex at:
;; https://github.com/nonsequitur/smex/

;; Another ofthen used mode is magit, which is an interface to git, allowing
;; you to manage your repos through emacs. You can read more about it at:
;; http://magit.vc/ It is one of the most useful modes available for emacs

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Section III: Global Key Bindings                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; By default C-x o is bound to 'other window, but I find I use it much more
;; ofther than open-line, which is bound to C-o, so I swap their definitions
(global-set-key (kbd "C-o") 'other-window)
(global-set-key (kbd "C-x o") 'open-line)

;; M-0..3 are bound to 'digit-argument. To be used with C-u. I don't use them
;; ofthen, so I prefer to rebind them to the window commands, since M-1 is
;; easyer to type than C-x 1.
(global-set-key (kbd "M-1") 'delete-other-windows)
(global-set-key (kbd "M-2") 'split-window-vertically)
(global-set-key (kbd "M-3") 'split-window-horizontally)
(global-set-key (kbd "M-0") 'delete-window)

;; Set the enter key to newline-and-indent which inserts a new line and then
;; indents according to the major mode. This is very convenient.
(global-set-key (kbd "<RET>") 'newline-and-indent)

;; If you have installed smex, you can uncomment the following lines. To
;; activate it without restarting Emacs select the lines and type M-x eval-regio\
n:

;; (require 'smex)

;; (global-set-key (kbd "M-x") 'smex)
;; (global-set-key (kbd "M-X") 'smex-major-mode-commands)
;; (global-set-key (kbd "C-c C-c M-x") 'execute-extended-command)
```

Slime

The Superior Lisp Interaction Mode for Emacs is the most widely used IDE for Common Lisp development. It augments Emacs with an editing mode, a REPL, a debugger and an object inspector, as well as referencing facility allowing you to jump to various points in Lisp source code.

Installation

The easiest way to install slime is to use Quicklisp and Roswell. From the command line do this:

```
 $ ros -Q -e '(ql:quickload :quicklisp-slime-helper)' -q
```

This line will tell roswell to start Lisp with Quicklisp loaded and evaluate the expression `(ql:quickload :quicklisp-slime-helper)` which installs Slime and then quit.

Slime has two parts, The emacs part called `slime` and the Common Lisp part called `swank`. In order for Emacs to know where slime is, we must add the following line to our `.emacs` file:

~/.emacs

```
(load (expand-file-name "~/.roswell/impls/ALL/ALL/quicklisp/slime-helper.el"))
```

Now we must tell slime how to start a Lisp repl with the following code

~/.emacs

```
(setf slime-lisp-implementations
      `((sbcl ("ros" "-Q" "-l" "~/.sbclrc" -L "sbcl" "run"))
        (ccl  ("ros" "-Q" "-l" "~/.ccl-init.lisp" "-L" "ccl-bin" "run"))))

(setf slime-default-lisp 'sbcl)

(setq slime-net-coding-system 'utf-8-unix)
```

The first form is a list of installed Lisp implementations and how to start them. By default Roswell doesn't load the implementation init files(`.sbclrc` and `.ccl-init.lisp` in this case) so we include them with the `-l` options. With the `-L` option we specify which implementation we want to start. The second form sets SBCL as the default. When you start slime this is the one that will be used. In order to start slime with a different implementation you could do `M-- M-x slime ccl` and a repl running ccl will start up. The last form tells slime to use `utf-8` when it's communicating with the Lisp process.

Next Let's modify the way buffers with Lisp code behave:

~/.emacs

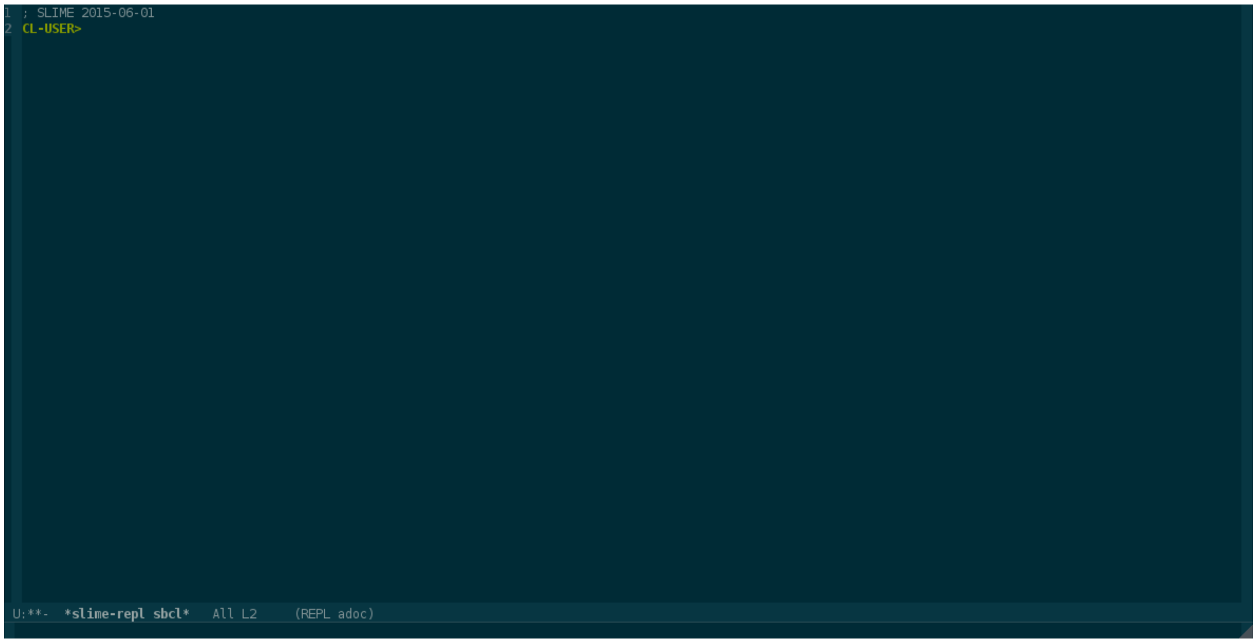
```
(defun lisp-hook-fn ()
  (interactive)
  ;; Start slime mode
  (slime-mode)
  ;; Some useful key-bindings
  (local-set-key [tab] 'slime-complete-symbol)
  (local-set-key (kbd "M-q") 'slime-reindent-defun)
  ;; We set the indent function. common-lisp-indent-function
  ;; will indent our code the right way
  (set (make-local-variable lisp-indent-function) 'common-lisp-indent-function)
  ;; We tell slime to not load failed compiled code
  (setq slime-load-failed-fasl 'never))

;; Finally we tell lisp-mode to run our function on startup
(add-hook 'lisp-mode-hook 'lisp-hook-fn)
```

The function `lisp-hook-fn` is set up to be called whenever `lisp-mode` is activated in emacs, it starts `slime-mode` sets up a few key bindings and alters a few settings. This code makes the `Tab` key use the slime auto-completion function instead of the default one. It also binds `M-q` to the `lisp-indent-function` Whenever you type `M-q` while the point is in a lisp block of code it will re-indent it according to the lisp convention. For that reason we set the `lisp-indent-function` to `common-lisp-indent-function`. We also tell slime to never load code that failed compilation.

The Slime repl

The Slime repl is one of it's most central features. You might have noticed that SBCL and CCL both have very simple repls, and are generally unsuited for interactive work. The slime repl on the other hand is much more feature complete. It starts up automatically when you type `M-x slime`.

This is how it looks like when you start it:



```
; SLIME 2015-06-01
CL-USER>
```

```
U:**-  *slime-repl sbcl*   All L2     (REPL adoc)
```

Slime repl

A new buffer called `*slime-repl sbcl*` is created where the repl lives. Notice the repl prompt is `CL-USER>`. This signifies that by default you are in the `CL-USER` package. If you change the package, for example with `in-package` the prompt will change to reflect that.

In the repl you have features like auto-completion bound to the `Tab` character, and a full history. You can move up and down the history with the arrow keys or with `M-p` and `M-n`. You can also search the history up and down with `M-s` and `M-r` which function very similarly to the Emacs built-in commands `C-s` and `C-r`.

Slime also displays the argument lists for functions and macros in the mini-buffer, for example if you type `(mapcar ` in the repl, the mini-buffer will display `(mapcar function list &rest more-lists)` telling you how to use the function.

Slime also includes a bunch of built in shortcuts you can activate by pressing the `,`(comma) key, when you press it you will be prompted in the mini-buffer for a command. One such a command is `!p` which sets the current package, for example `, !p foo-package` will set the current package to `foo-package`. Note that the package must be defined. You also get auto-complete with `Tab` in this prompt. For a complete list of commands see here[21]

Editing Lisp code

Paredit

TBW

Navigation

Slime includes a function to go to the definition of a symbol and edit it called `slime-edit-definition`, by default it's bound to `M-.`. You can use it as many times as you wish, and when you are done you can go back to the previous location you were editing with the command `slime-pop-find-definition-stack` bound to `M-,`.

Compilation and evaluation.

When you are editing lisp code you would want to run your code. There are several ways to do this. The first one you already know about, you can use `ql:quickload` to load your project into Lisp, but while you're editing code you'll want to re-compile individual files or even individual code blocks. Slime includes functions to help you with this.

First let's talk briefly about running lisp code. There are two ways you can run lisp code, the first is to evaluate it, the second is to compile and load it. When you tell slime to evaluate your code it sends it to Lisp and calls `eval` on it. Depending on how `eval` is implemented your implementation might directly interpret the code, or it might compile it first. For example by default SBCL compiles all code unless you enable the interpreter explicitly. When you tell slime to compile your code it sends it to Lisp and runs `compile` on it.

Here are some of the available evaluation commands:

- `C-x C-e` will evaluate the last expression before the point and display the result in the mini-buffer.
- `C-M-x` will evaluate the top-level form where the point is in, for example if you're in the middle of editing a `defun` form it will evaluate it. Again the result will be displayed in the mini-buffer.
- `C-c C-r` will do the same to a selected region.

[21] https://www.common-lisp.net/project/slime/doc/html/Shortcuts.html#Shortcuts

Generally you would use these commands if you want to see the result of evaluation in the mini-buffer. For example evaluating `(+ 1 2)` would display `3`, but compiling it will simply tell you that compilation was finished and not the result.

Additionally another useful evaluation command is `C-c C-j` it evaluates the last expression by copying it to the repl and displaying the result there.

Compilation works slightly differently. When your code is compiled and run, you will only see a “compilation finished” message as well as a summary of how many notes, warnings and errors you got.

The most common compilation commands are:

- `C-c C-c` compiled the top-level form. It works similarly to `C-M-x`. You’ll be using this one the most often when working with lisp code.
- `C-c C-k` compiles the whole file.

Generally when you’re working with lisp code you will be recompiling functions as you edit them and testing the results in the repl. When you compile code you get the result of the compilation printed in the `*slime-compilation*` buffer. This is where all of the compiler notes(errors and warnings) will be displayed. Let’s look at an example, suppose I have a file with the following lisp *intentionally* wrong function:

foo.lisp

```
(defun foo (foo)
  (let (nil)))
```

Now I put the point anywhere in that form and type `C-c C-c` the `*slime-compilation*` buffer will pop-up with the following contents:

slime-compilation

```
cd /home/pav/example/
1 compiler notes:

foo.lisp:1:1:
  style-warning:
    The variable FOO is defined but never used.
    --> PROGN SB-IMPL::%DEFUN SB-IMPL::%DEFUN SB-INT:NAMED-LAMBDA
    ==>
      #'(SB-INT:NAMED-LAMBDA FOO
            (FOO)
          (BLOCK FOO
            (LET (())
              )))



foo.lisp:2:3:
  error: NIL names a defined constant, and cannot be used as a local variable.

Compilation failed.
```

We see 1 style warning and 1 error. The style warning states that the variable `foo` is never used even though we defined it. This is valid code, but could be a sign of a bug. The error states that the symbol `NIL` cannot be used as a local variable in `let`. Notice the warnings and notes also include the file name and the position, those lines are links to the file position where the note occurred. For example you could click with the mouse on them and they will bring you to the offending code.

The compilation buffer includes custom keyboard commands to navigate it. The commands `M-p` and `M-n` will go through all of the notes. If you typed `C-c C-c` in this buffer it will bring you to the offending code as if you clicked on the link. Notice also in the code buffer that the wrong code is marked with colored underlines: Yellow for a warning and red for an error. Once you're done with the compiler buffer you can quit it with the `q` key.

The Slime debugger

TBW

The Slime inspector

Slime includes an inspector witch which you can inspect values in your running system. Suppose we have the following code:

foo.lisp

```lisp
(defparameter *list* (list "Hello" 'world))
```

Our list contains a string and a symbol. If you compiled this code you can inspect the variable by moving the point to the name `*list*` and typing `C-c I`. The inspector buffer will pop up and it will look something like this:

```
#<CONS {10037829B7}>
--------------------
A proper list:
0: "Hello"
1: WORLD
```

This tells you that the object you're inspecting is a cons cell and is a proper list, followed by it's contents. You can use the `Tab` key to move between the inspectable values in the buffer and descent into them. For example if you type `Tab` the point will move to the value `"Hello"` if you now type `Return` you will inspect that value. It will look like this:

```
#<(SIMPLE-ARRAY CHARACTER (5)) {100378293F}>
--------------------
Dimensions: (5)
Element type: CHARACTER
Total size: 5
Adjustable: NIL
Fill pointer: NIL
Contents:
0: #\H
1: #\e
2: #\l
3: #\l
4: #\o
```

This tells us that the string is a simple array of characters, it has `5` elements and we can do the same again. Type `Tab` until the point is on the `H` character and press `Return`:

```
#<STANDARD-CHAR {4849}>
--------------------
Char code: 72
Lower cased: #\h
Upper cased: @2=#\H
```

You can go on like this, but how do you go back? The `l` key will pop you back up. Pressing it a couple of times will get you back to the inspector for `*list*`. You can quit the inspector with `q`. ### Other useful information

Here are some other useful features you might want to know about:

- `C-c C-d d` will describe a symbol. A buffer with a description of the symbol will pop-up.
- `C-c C-d f` will do the same but for functions.

- `C-c C-d a` will run `slime-apropos`, it will search the defined symbols with the string you supply it.
- `C-c C-d h` will look-up a symbol in the hyperspec and display the result in a web browser. You'll be using this one a lot.
- `C-c C-m` and `C-c M-m` are bound to `slime-macroexpand-1` and `slime-macroexpand-all` respectively. A buffer will open with the macro expansion.

Sly

TBW