;;;;;;;;;;;;;;;;;;;;;;;;

;;;; Lisp Web Tales ;;;;

;;;;;;;;;;;;;;;;;;;;;;;;

;; My attempts at learning web development,
;; foolishly using Common Lisp,
;; and even more foolishly,
;; writing about it in public

;;;; A book By Pavel Penev

```
(restas:define-module #:lisp-web-tales
    (:use #:cl #:restas))

(in-package #:lisp-web-tales)

(define-route about ("about/" :method :get)
  '(:features
    ("Simple tutorials using various lisp libraries"
     "An in-depth look into some of them"
     "My irrelevant oppinions on Lisp and the web"
     "Common Lisp cult propaganda for the unenlightened")
    :target-audience
    ("Anyone interested in Lisp"
     "People who find Node.js too mainstream")))
```

Lisp Web Tales

My attempts at learning web development, foolishly using common lisp, and even more foolishly, writing about it in public

Pavel Penev

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

This version was published on 2013-11-24

Tweet This Book!

Contents

Preface

I am an enthusiast if there was ever such a thing. So this is an enthusiasts book, written out of joy and curiosity, and as an escapist pleasure in a time when the outside world is closing in on me, and my time for lisp is running short. Exams, graduation, the eventual job search, and employment as a Blub coder is what is in front of me for 2013.

To me Lisp is one of the most fun and easy to use languages out there, it has challenged me intellectually, and provoked my thinking in all sorts of directions, from basic software design, to how software communities work. All of these questions have led me to believe that the right course for me personally is to continue to learn Common Lisp and its history. I will not be a worse programmer if I continue to invest effort into mastering it, just the opposite. The same is true for all sorts of languages and platforms, and some of them I am also investing my self in, such as GNU Emacs, Linux, and as horribly flawed as it is, the web. Whatever my day jobs might be in the future, I will continue my hobbyist practice as a programmer, and until I find a better tool, I will continue to use and love Common Lisp.

This book is in a way an attempt at maintaining that practice and getting my skill level up. It has taken a lot of research and experimentation, and helped me improve my writing. So even if it fails to attract an audience, and even if left unfinished, it is well worth the effort.

Pavel Penev, March 2013

Introduction

Why Lisp

Today we have more programming languages than we can count. Somehow, Lisp still manages to stand out, at least for me. I've been obsessed with the lisp family of languages for four years now, and I've been especially interested in Common Lisp, which I consider to be the best general purpose dialect. It is an easy language to pick up, and a difficult language to master. So far, every day spend learning lisp has been a huge plus for me, so all those difficulties have been worth it. Lisp is fun, challenging and rewarding of such efforts. No language I've picked up since or before has felt the same way, they were all either needlessly complex(most of the complexity in lisp is there for a reason), or too simplistic and lacking in sophistication when that is needed.

As for how practical this language is for web development, It's as practical as you make it. Lisp is the perfect language for the gray areas where were we still haven't quite figured out how to do things. I believe the web is one such area, and experimentation and playful exploration of ideas is vital. This is what Lisp was designed for, not for the web specifically, but for what it is, a new playground where flexibility and creativity have room to grow.

Common Lisp has been a faithful ally in my self-education. Maybe it can be one for you too.

Whats in the book

The book is a set of tutorials and examples. It uses the Common Lisp language and some of the libraries we'll be using for the examples and tutorials include:

- The Hunchentoot web server
- The Restas web framework
- The SEXML library for outputting XML and HTML
- Closure-template for HTML templating
- Postmodern for PostgreSQL access, and cl-reddis as a simple datastore
- Various utilities

Who is this for

This book is for anyone interested in Lisp and web apps. I assume you have some familiarity with both subjects, but I don't assume you are a Lisp expert, you can just read a few tutorials to get the basics and get back to my book to get started with web apps. I've linked some of them in Appendix B. So you need to know what `(+ a b)` means, I won't explain html and css to you, and HTTP shouldn't be a scary mystical acronym to you. Also some knowledge of databases would be good. In other words, I assume you are a programmer, know the basics and just want to play around with Lisp.

What you need to get started

A lisp implementation, preferably sbcl(recommended for Linux users) or ccl(recommended for Mac and Windows users), and Quicklisp, the Common Lisp package manager. I've written a quick "getting started" tutorial in Appendix A. And the links in Appendix B have additional information.

You will also need a text editor which supports matching parenthesis, so no notepad. Appendix A has some recommendations, but the best way to use Lisp is with Emacs and the Slime environment. A similar environment is available for Vim users with the Slimv plugin. If you don't already know Emacs or Vim, you can leave learning it for later, and just use any old code editor and the command line. If you are serious about Lisp though, consider picking up Emacs eventually.

Appendix B also has a lot of links you can go to to find more about Lisp, including tutorials, books, wikis and places you can ask your questions.

Typographic conventions

Inline code:

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, => 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:\
```

1 The basics

Raw example

Here is a complete hello-world web application, saved in the file `hello-world.lisp`:

```
;;;; hello-world.lisp

(ql:quickload "restas")

(restas:define-module #:hello-world
    (:use :cl :restas))

(in-package #:hello-world)

(define-route hello-world ("")
  "Hello World")

(start '#:hello-world :port 8080)
```

This apps basically returns a page with the text “hello world” to any request at the “/” uri. It can be run from the command line using sbcl or ccl like this:

```
$ sbcl --load hello-world.lisp
```

or

```
$ ccl --load hello-world.lisp
```

Or loaded from the lisp prompt:

```
 * (load "hello-world.lisp")
```

Now you can open the page http://localhost:8080/[1] and see the result.

Detailed explanation

I’ll do an almost line by line explanation of what is happening.

[1]http://localhost:8080/

```
(ql:quickload "restas")
```

All examples in this book will be using the hunchentoot web server, and the RESTAS web framework built on top of it.

As you can read in the Appendix A, the way we install and load libraries with Quicklisp is with the `quickload` function. The `ql:` part simply means that the function is in the ql package, which is a short name for the `quicklisp` package. Lisp packages often have such short alternative names, called nicknames. This line simply loads Restas, and installs it if it isn't already present. Since hunchentoot is a dependency for Restas, it gets loaded as well.

```
(restas:define-module #:hello-world
    (:use :cl :restas))

(in-package #:hello-world)
```

Restas applications live in modules, which are similar to ordinary common lisp packages(and in fact, a package is being generated behind the scenes for us), we define them with the macro `define-module` from the `restas` package. It has the same syntax as common lisps `defpackage`. We give our module the name `hello-world` and specify that we want all public symbols in the `cl` and `restas` packages to be imported into our module. We then set the current package to `hello-world`. All the code after this form to the end of the file will be in that package.

Symbols starting with `#:` are uninterned, meaning they have no package, we just want to use its namestring, which is `"HELLO-WORLD"`. Uninterned symbols are useful if you want a lightweight string to name something, in this case a package.

The following form `(:use :cl :restas)` means that all the "public" symbols from the packages `cl`(a standard package containing all lisp functions, variables, classes etc) and `restas` get imported into our `hello-world` package, so we don't have to write `restas:define-route` and can simply say `define-route`.

```
(define-route hello-world ("")
  "Hello World")
```

Restas apps are based on uri handlers called routes. Routes in their simplest form shown here, have: * A name (`hello-world` in this case) * An uri template. in this case the empty string `""`, meaning it will match the `/` uri * A body generating a response, in this case the string "hello world" returned to the client.

There are a few more details to routes, but we'll get to them in a bit.

```
(start '#:hello-world :port 8080)
```

The Restas function `start` is used to initializes a module, and starts a hunchentoot web server. As a first argument we give it the symbol naming our module with our application defined in it and pass a port number as a keyword parameter. Note that the symbol must be quoted ith a `'`. Again, there is quite a bit more to this function, but for now, we just need to get our app running.

A simple blog

Lets look at a bit more complicated example: a simple blog app. It will be self contained in a single file you can run from the command line, just like the previous example. Subsequent examples will use ASDF and Quicklisp. In addition to Restas and Hunchentoot we'll also be using the SEXML library for html generation. The blog posts will be stored in memory as a list. The basic features would be:

- View all blog posts on the front page
- Separate pages for each post
- Separate pages for authors, listing all of their posts.
- Admin form for adding posts, protected by crude HTTP authorization.

The source code

Here is the complete source of our app, consisting of slightly over 100 lines of code:

```lisp
;;;; blogdemo.lisp

;;;; Initialization

(ql:quickload '("restas" "sexml"))

(restas:define-module #:blogdemo
  (:use #:cl #:restas))

(in-package #:blogdemo)

(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))

(<:augment-with-doctype "html" "")

(defparameter *posts* nil)

;;;; utility

(defun slug (string)
  (substitute #\- #\Space
              (string-downcase
               (string-trim '(#\Space #\Tab #\Newline) string))))

```

```lisp
;;;; HTML templates

(defun html-frame (title body)
  (<:html
   (<:head (<:title title))
   (<:body
    (<:a :href (genurl 'home) (<:h1 title))
    body)))

(defun render-post (post)
  (list (<:div
         (<:h2 (<:a
                :href (genurl 'post :id (position post *posts* :test #'equal))
                (getf post :title)))
         (<:h3 (<:a
                :href (genurl 'author :id (getf post :author-id))
                "By " (getf post :author)))
         (<:p (getf post :content)))
        (<:hr)))

(defun render-posts (posts)
  (mapcar #'render-post posts))

(defun blogpage (&optional (posts *posts*))
  (html-frame
   "Restas Blogdemo"
   (<:div
    (<:a :href (genurl 'add) "Add a blog post")
    (<:hr)
    (render-posts posts))))

(defun add-post-form ()
  (html-frame
   "Restas Blogdemo"
   (<:form :action (genurl 'add/post) :method "post"
    "Author name:" (<:br)
    (<:input :type "text" :name "author")(<:br)
    "Title:" (<:br)
    (<:input :type "text" :name "title") (<:br)
    "Content:" (<:br)
    (<:textarea :name "content" :rows 15 :cols 80) (<:br)
    (<:input :type "submit" :value "Submit"))))

;;;; Routes definition

```

```
(define-route home ("")
  (blogpage))

(define-route post ("post/:id")
  (let* ((id (parse-integer id :junk-allowed t))
         (post (elt *posts* id)))
    (blogpage (list post))))

(define-route author ("author/:id")
  (let ((posts (loop for post in *posts*
                  if (equal id (getf post :author-id))
                  collect post)))
    (blogpage posts)))

(define-route add ("add")
  (multiple-value-bind (username password) (hunchentoot:authorization)
    (if (and (equalp username "user")
             (equalp password "pass"))
        (add-post-form)
        (hunchentoot:require-authorization))))

(define-route add/post ("add" :method :post)
  (let ((author (hunchentoot:post-parameter "author"))
        (title (hunchentoot:post-parameter "title"))
        (content (hunchentoot:post-parameter "content")))
    (push (list :author author
                :author-id (slug author)
                :title title
                :content content) *posts*)
    (redirect 'home)))

;;;; start

(start '#:blogdemo :port 8080)
```

This file can be run from the command line like so:

```
$ sbcl --load blogdemo.lisp
```

or

```
$ ccl --load blogdemo.lisp
```

Or load it from the Lisp prompt:

```
 * (load "blogdemo.lisp")
```

The username and password for adding new posts, as can be seen in the source, are “user” and “pass” respectively. Try adding posts, and vary the names of authors. Explore how the app behaves. In later chapters we will learn how to improve it a bit, but for now, it will do.

Source walk-through

Lets walk through the various sections of this source code and see how it works.

Initialization

```
(ql:quickload '("restas" "sexml"))
```

We begin by loading the libraries we’ll be using: Restas and sexml.

```
(restas:define-module #:blogdemo
  (:use #:cl #:restas))

(in-package #:blogdemo)
```

This time our application is named `blogdemo`.

```
(sexml:with-compiletime-active-layers
    (sexml:standard-sexml sexml:xml-doctype)
  (sexml:support-dtd
   (merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))
   :<))

(<:augment-with-doctype "html" "")
```

SEXML is a library for outputting XML using lisp s-expressions as input. It takes an xml dtd and generates a package with functions for all the necessary tags. In our case, we give it an html5 dtd, and specify the package named `<`. This means that we can write code like:

```
(<:p "Hello world")
```

and get this out:

```
<p>Hello world</p>
```

A thing to note is that SEXML comes with an html5 dtd file as part of the distribution. The code `(merge-pathnames "html5.dtd" (asdf:system-source-directory "sexml"))` is used to find the path to that file. Don’t worry about how this actually works, essentially it means “give me the path to the file ‘html5.dtd’ in the sexml installation directory”.

And finally, we define our “database” as an empty list named by the variable `*posts*`:

```
(defparameter *posts* nil)
```

Utility

I've included a section for utility functions, which at this point contains only one function:

```
(defun slug (string)
  (substitute #\- #\Space
              (string-downcase
               (string-trim '(#\Space #\Tab #\Newline)
                            string))))
```

If you are familiar with Django, you probably know the term 'slug'. A slug is a string we can use in urls. The `slug` function takes a string, such as `" Foo Bar BaZ "` and converts it to a url friendly string like `"foo-bar-baz"` by trimming surrounding white space, converting all the characters to lower case and substituting the spaces between words for dashes. We'll be using it to create ID's for authors in our "database".

HTML templates

In general the rules for using sexml for html generation are as follows:

```
(<:tagname attributes* content*)
```

where attributes can be of the form:

```
:key value
```

and the content can be a string or a list of strings to be inserted into the body of the tag. For example, this snippet:

```
(<:a :href "/foo/bar" "This is a link to /foo/bar")
```

Will produce the following HTML: `<a href="/foo/bar">This is a link to /foo/bar</a>`

Lets take a look at the various template functions we'll be using:

```lisp
;;;; HTML templates

(defun html-frame (title body)
  (<:html
   (<:head (<:title title))
   (<:body
    (<:a :href (genurl 'home) (<:h1 title))
    body)))
```

`html-frame` is a function, which takes a title and a body and converts it to an html page whose body has a link to the home page at the top. We can call it like so:

```lisp
(html-frame "This is a title" "This is a body")
```

And get the following output as a lisp string(I've indented it, and broken it up to separate lines):

```html
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <a href="/"><h1>This is a title</h1></a>
    This is a body
  </body>
</html>
```

Of note is the use of the restas function `genurl` which takes a route, and generates a url that would be handled by the route function. In this case `(genurl 'home)` will generate the `/` url, since that is what the `home` route(defined in the next section) handles. This is done because restas applications can me "mounted" on different points in a url path tree. if the whole application is mounted on the uri `/blog`, then the same code(without having to change it) would generate `/blog/` as the output.

Before we look at how we generate the blog posts themselves, let me explain how we store them. We store blog posts as a list of plists, a convention for lists where the odd numbered elements are keys, and the even numbered elements are values. Plists are useful as lightweight maps, and look like this:

```lisp
'(:author "Author Name"
  :author "author-name" ; this is a slug string
  :title "This is a title"
  :content "This is the body of the blog post")
```

By convention, keys in plists are common lisp keywords, starting with colons. Elements in a plist can be accessed with the function `getf`, which takes a plist and a key as it's argument. So if we wanted to get the name of the author from the plist `post`, we would write `(getf post :author)`. Simple as that. Now lets look at how we use them:

```
(defun render-post (post)
  (list (<:div
         (<:h2 (<:a
                :href (genurl 'post :id (position post *posts* :test #'equal))
                (getf post :title)))
         (<:h3 (<:a
                :href (genurl 'author :id (getf post :author-id))
                "By " (getf post :author)))
         (<:p (getf post :content)))
        (<:hr)))

(defun render-posts (posts)
  (mapcar #'render-post posts))
```

The function `render-post` takes a blog post and renders it as html. The `genurl`'s in this function are a bit more complicated. In this case `genurl` has to generate urls to each individual post, which requires additional information, such as it's ID. We use the posts position is the list of posts as it's id, so each post would have a url like `posts/1` or whatever it's number in the list is. Same is true for the author, except authors are identified by a slug of their name. so the url would look like `author/author-name`. This works because routes can handle more than one url with similar structure, for instance both `posts/1` and `posts/2` will be handled by the route `post`, We'll see how that works in a minute.

The function `render-posts` simply takes a list of posts, and renders each one individually, into a list of html strings. It uses the `mapcar` function, which might be called `map` or `each` in other languages.

```
(defun blogpage (&optional (posts *posts*))
  (html-frame
   "Restas Blogdemo"
   (<:div
    (<:a :href (genurl 'add) "Add a blog post")
    (<:hr)
    (render-posts posts))))
```

`blogpage` takes a bunch of blog posts and renders a complete html page with them. By default it renders all of the posts, but we can give it a subset, as we do when we show only the posts by one author.

And finally, `add-post-form` generates a page with an html form in it for adding a blog post:

```
(defun add-post-form ()
  (html-frame
   "Restas Blogdemo"
   (<:form :action (genurl 'add/post) :method "post"
    "Author name:" (<:br)
    (<:input :type "text" :name "author")(<:br)
    "Title:" (<:br)
    (<:input :type "text" :name "title") (<:br)
    "Content:" (<:br)
    (<:textarea :name "content" :rows 15 :cols 80) (<:br)
    (<:input :type "submit" :value "Submit"))))
```

This is it for html generation.

Routes

Route handlers are the heart of any Restas application. The complete syntax for `define-route` is:

```
(define-route name (template &key method content-type)
  declarations*
  body*)
```

We've seen a very basic usage of this. The blog example doesn't use the optional declarations, we'll cover them later, but the optional method keyword parameter will come in handy when we need to handle `POST` data from a form. By default it's value is `:get`, but can be any HTTP method. Using this we can have routes with the same uri template, but different HTTP methods, this makes Restas very well suited for RESTful APIs as the name suggests. Let's take a look at the individual routes:

```
(define-route home ("")
  (blogpage))
```

Simply displays the home page.

```
(define-route post ("post/:id")
  (let* ((id (parse-integer id :junk-allowed t))
         (post (elt *posts* id)))
    (blogpage (list post))))
```

The `post` route handles the case where we are viewing a single blog post. We see that the `post` route has an interesting template: `"post/:id"`. If you are familiar with something like Sinatra, you'll find this syntax familiar. Parts of a uri template beginning with a colon designate route variables, and can match any uri with that structure, as I mentioned in the previous section. For example `/post/0`, `/post/1` and `/post/2` will all match and be handled by this route. But so will `/post/foo`, our app breaks if we go to a url that doesn't have an integer url component. We'll see later how we can fix this, for now, we simply don't do that. The matched string also gets bound to a lisp variable in the body of the route, in our case `id`.

Lets look at the body, each such route template variable is bound to the string that it matched, so we get values like `"0"`, or `"1"` or `"foo"`. Using common lisps `parse-integen` we convert it to an integer, and we look up the element in the list of posts with the `elt` function, which takes a list(or any lisp sequence) and gets the item at the index we suply as a second argument.

We render the post by passing it as a list to `blogpage`, which returns a string, which in turn, Restas returns to the browser.

```
(define-route author ("author/:id")
  (let ((posts (loop for post in *posts*
                  if (equal id (getf post :author-id))
                  collect post)))
    (blogpage posts)))
```

The `author` route is very similar. we have an `:id` variable as well, but it can be any string, so we don't worry about parsing it. We use common lisps powerful loop macro to iterate over all the posts, and if the id we supply in the url matches the `:author-ids` of the individual posts, we collect them into a new list. `:author-id` is generated as a slug version of the author name, specifically so that we can use it as a key and as a part of a url.

If we have blog posts by an author named `"Pavel Penev"`, its slug would have been saved it into the database as `"pavel-penev"`, and if we go to the uri `author/pavel-penev`, we'll see all the posts by that author on the page.

```
(define-route add ("add")
  (multiple-value-bind (username password) (hunchentoot:authorization)
    (if (and (equalp username "user")
             (equalp password "pass"))
        (add-post-form)
        (hunchentoot:require-authorization))))
```

The `add` route handles displaying a form for the user to submit a blog post. Since we don't want just anybody to add posts, we want to add some user authentication, but since this is just a simple example, I won't bother with login forms, cookies and sessions, we'll leave that for a later chapter. For now I'll use simple HTTP authorization.

If you are unfamiliar with HTTP authentication, it is a very crude way to log into a web site. The browser has to supply a username and a password as an HTTP header. The function `hunchentoot:authorization` returns them as two separate values, since common lisp supports multiple return values, instead of just one(as is the case in probably every other language you've ever used), we have to bind them using the macro `multiple-value-bind`, which is like `let` for multiple values. It binds the variables `username` and `password` and in the body we check if they are the expected values, in our case "user" and "pass". If they are, we render our form, and if not, we tell the browser to ask the user for a username and password using 'hunchentoot:require-authorization'.

```
(define-route add/post ("add" :method :post)
  (let ((author (hunchentoot:post-parameter "author"))
        (title (hunchentoot:post-parameter "title"))
        (content (hunchentoot:post-parameter "content")))
    (push (list :author author
                :author-id (slug author)
                :title title
                :content content) *posts*)
    (redirect 'home)))
```

Finally, `add/post` handles the data send to us by the form generated by `add`. We specify that the http method should be `POST`, and use the function `hunchentoot:post-parameter` to extract the user submitted values from the request. The strings `"author"`, `"title"` and `"content"` were the names of fields in our form. We bind them to values, and built a plist using the function `list`. Note that we add the `:author-id` key, with a value generated by applying `slug` to the `author` string. The list we `push` onto the database variable `*posts*`. `push` takes an item, and adds it to the front of a list. At the end we `redirect` back to the home page. `redirect` is a restas function with much the same syntax as `genurl` but instead of generating a url out of a route name, it generates the url, and tells the browser to redirect to it.

Conclusion

This concludes the honeymoon chapter. We saw all the very basic ideas: A restas web application is a module, which is a collection of route functions that handle uri requests. There is quite a bit more to it than that, and we'll get to it in the next chapters. The code was kind of bare bones, usually we would like to have an ASDF system defined, so we can have all the lisp infrastructure available for us (have Quicklisp download all of our dependencies, have our templates be compiled automatically, and be able to develop interactively). At the moment our apps are bare scripts, and lisp is not a scripting language, even though you can use it as such. It's a powerful interactive system, and we would be fools if we didn't take advantage of that power.

Appendix A: Getting started

Linux

Getting a Lisp implementation

The two implementations I recommend for use in this book are SBCL and CCL, both are very good, open source and generate fast code. If you are on windows or OS X, I recommend CCL, SBCL on Linux. I've had at least two people report to me problems with sbcl on OS X and and my tutorials, which is probably because of improperly built binaries, rather than an actual problem, but if you don't feel like compiling your SBCL from source(which I recommend), stick with CCL on those platforms for now.

Most Linux distributions have both CCL and SBCL in their package repositories, for example on Debian derived systems such as Ubuntu you can install sbcl with apt-get:

```
$ sudo apt-get install sbcl
```

But I recommend you download and install binaries manually, distributions sometimes patch CL implementations in order to "fix" something. Also who knows how ancient the version in the package manager is. It is usually recommended to work with the latest releases of CL implementations.

SBCL

You can download SBCL at http://www.sbcl.org/platform-table.html

Once you've done so, uncompress the archive. The example is shown for x86-64 on Linux:

```
$ tar -jxvf sbcl-1.1.5-x86-64-linux-binary.tar.bz2
```

Go to the directory:

```
$ cd sbcl-1.1.5-x86-64-linux/
```

The file `INSTALL` has information about how to configure the installation, but the default should suit your needs just fine, type:

```
$ sh install.sh
```

type `sbcl` into the command line to see if it works OK. you should get a prompt starting with an `*`. I have the habit of typeing `(+ 1 2)` in order to see if it really works, and I have never gotten an answer different than 3 so far, that's reliable software :)

CCL

You can get CCL from http://ccl.clozure.com/download.html The distribution contains both the 64 and 32 bit binaries. Chapter 2[2] of the CCL manual contains information on how to obtain and install CCL if you need it.

After you download CCL, uncompressed the archive with the following command:

```
$ tar -xzvf ccl-1.8-linuxx86.tar.gz
```

CCL is started by a shell script in the `ccl/scripts` directory, named `ccl` or `ccl64` for the 32 and 64 bit versions respectively. The way you install CCL is by copying one(or both) of these scripts to a directory on your path, and editing them to point to the CCL directory you just uncompressed. so for example if my `ccl` directory is in my home directory, named `/home/pav` on Linux:

```
$ sudo cp /home/pav/ccl/scripts/ccl64 /usr/local/bin
```

I then edit it to point to the `ccl` directory by setting the value of the variable `CCL_DEFAULT_DIRECTORY` at the top of the file to the `/home/pav/ccl/`.

Since I don't use the 32 bit version, I rename the file to simply `ccl`

```
$ sudo mv /usr/local/bin/ccl64 /usr/local/bin/ccl
```

I then ensure the file is executable:

```
$ sudo chmod +x /usr/local/bin/ccl
```

type `ccl` at the command line. The prompt should be `?`. Type some expression like `(+ 1 2)` to see if it works.

Installing Quicklisp

Quicklisp is a package manager for lisp. It handles downloading and installation of libraries. Installing it is rather easy. More information and documentation can be found at http://www.quicklisp.org/beta/

Download the file http://beta.quicklisp.org/quicklisp.lisp

Load it with sbcl or ccl:

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

This will load the file into lisp and we can proceed to install it. Type the following into the lisp prompt:

[2] http://ccl.clozure.com/manual/chapter2.html

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

This will install quicklisp in your home directory.

In order to make sure quicklisp is loaded every time you start lisp, type the following:

```
(ql:add-to-init-file)
```

And you're done. You can now quickload libraries, for instance the following command will install the Restas web framework:

```
(ql:quickload "restas")
```

Recommended editors

- Emacs and Slime: The best option if you already know it, or you are willing to learn it.
- Vim and Slimv: The next best thing. Vim isn't actually easier to learn than Emacs, but if you already know it, it can get the job done.
- All the other options pretty much stink, but Kate at least has a built in terminal, so it's a bit easier to work with lisp interactively.

Windows

Getting a Lisp implementation

The implementation I recommend on Windows is CCL, you can download it from here[3].

After you've downloaded the file, uncompress it in the directory `C:\ccl`.

The `ccl` folder will have two executables, one named `wx86cl` for 32 bit systems, and `wx86cl64` for 64 bin systems.

At the command prompt, we can start the application by typing:

```
 > c:\ccl\wx86cl
```

Let's make it possible to start ccl simply by typing `ccl`. I'll demonstrate for the 32 bit version, it is equivalent for the 64 bit.

First, rename the `wx86cl` and `wx86cl.image` files to `ccl` and `ccl.image` respectively. Now, we need to set up the PATH enviromental variable so that windows knows where to find CCL.

For Windows 7, click the Start menu, and right click on `Computer` and select `properties`. From the sidebar select `Advanced system settings`. At the bottom of the window, click on the `Environment Variables` button. In the second pane, called `System variables`, search for the `Path` variable, select it, click on `Edit`. At The end of the `Variable value` field, append the following: `;C\ccl\`. Click OK. Open a command prompt, and type ccl, it should greet you with a message. That's it, you have CCL installed.

[3]http://ccl.clozure.com/download.html

Installing Quicklisp

Quicklisp is a package manager for lisp. It handles downloading and installation of libraries. Installing it is rather easy. More information and documentation can be found at http://www.quicklisp.org/beta/

Download the file http://beta.quicklisp.org/quicklisp.lisp

Open a command prompt, and go to the directory where you downloaded it:

```
 > chdir path\to\download\directory
```

Load it with ccl:

```
 > ccl --load quicklisp.lisp
```

This will load the file into lisp and we can proceed to install it. Type the following into the lisp prompt:

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

This will install quicklisp in your home directory.

In order to make sure quicklisp is loaded every time you start lisp, type the following:

```
(ql:add-to-init-file)
```

You can now install lisp libraries using the `ql:quickload` command. Note that some libraries we'll be using depend on haveing OpenSSL installed, so make sure you install it, a third party installer is available from here[4]

Restart CCL, to test if it worked:

```
 ? (quit)
 > ccl
 ? (ql:quickload "restas")
```

If it started downloading and installing Restas, you're done. You can now quickload libraries.

Recommended editors

- Emacs and Slime: The best option if you already know it, or you are willing to learn it.
- Lisp Cabinet: A bundle of Emacs and various lisp implementations, an easy way to install Lisp and Emacs, with various customizations.
- Vim and Slimv: The next best thing. Vim isn't actually easier to learn than Emacs, but if you already know it, it can get the job done.
- Sublime Text2: Seems to be acceptable for editing lisp code.
- LispIDE: Barely qualifies as an IDE, but is an option you can look into.
- Notepad++: Popular code editor for Windows. Minimally acceptable as a lisp editor.

[4]http://slproweb.com/products/Win32OpenSSL.html

Appendix B: Recomended reading

Online tutorials

A lisp tutorial I like is Lisp in small parts[5]

If you are new to programming, people usually recommend the free book Common Lisp: A Gentle Introduction to Symbolic Computation[6].

For experienced hackers new to lisp, Practical Common Lisp[7] is probably the best way to learn the language.

Cliki: The Common Lisp wiki

Almost all information you would want about Common Lisp can be found on Cliki[8].

Cliki pages of note: Getting started[9]

Online tutorials[10]

Recomended libraries[11]

IRC

I(and many lispers) hang out on irc on the Freenode[12] server. Channels I frequent include `#lispweb` and `#lisp`. You can also find help on `#clnoobs`.

Check out a bunch of other lisp-related channels on cliki[13].

[5] http://lisp.plasticki.com/
[6] http://www-2.cs.cmu.edu/~dst/LispBook/
[7] http://www.gigamonkeys.com/book/
[8] http://cliki.net
[9] http://www.cliki.net/Getting%20Started
[10] http://www.cliki.net/Online%20Tutorial
[11] http://www.cliki.net/Current%20recommended%20libraries
[12] http://freenode.net/
[13] http://www.cliki.net/IRC