Lisp for the web

by adam Tornhill

Lisp for the Web

Adam Tornhill

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

This version was published on 2015-05-24

Tweet This Book!

Please help Adam Tornhill by spreading the word about this book on Twitter!

The suggested hashtag for this book is #lispforweb.

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=#lispforweb

Contents

Introduction

Lisp has kept me fascinated since I hesitantly opened my first pair of parenthesis. To this day, the lessons I took away from Lisp keep informing my decisions in other languages and technologies. Learning Lisp radially changed how I view programming.

But starting with Lisp wasn't a smooth transition. Lisp's power comes from a philosophy that's absent in all mainstream languages of today. In a way, learning Lisp was like facing programming as an absolute beginner again.

When it comes to learning, there's nothing more efficient than teaching someone else. Suddenly your cloudy thoughts need to be crystalized, put in context and communicated. In short, you need to truly understand your topic. Following that approach I set out to write an article on Lisp programming. I chose the web as my domain since my preliminary goal was to understand how a 50 year old language could provide just the right tools for an environment that wasn't even conceived back then. If Lisp indeed made it, there had to be something timeless over it. Some inherent quality that could give the programmers of today unequalled power and flexibility.

Writing the original Lisp for the Web article back in 2008 proved to be a rewarding experience. Not only did I succeed on this step in my learning journey; the feedback I got from fellow programmers around the world was overwhelming. I never thought I'd reach such a wide audience. In a way Lisp for the Web seemed to provide a natural next step for people that read Paul Graham's essays and were now looking for practical material to take their first step into the exciting (and sometimes frightening) world of Lisp.

Since that time back in 2008 there has been several additions to the Common Lisp ecosystem. Quicklisp lowered the hurdle by providing a simple way to get started, making all powerful libraries easy to install. New books turned up. In particular Land of Lisp[1] stands as an excellent example on a pedagogical introduction to the language. The web development community continued to evolve too. Just to mention a few examples, Edi Weitz made several improvements to his Hunchentoot web-server, NoSQL backends gained popularity, and Matthew Snyder expanded on my work with his excellent Lisp for the Web, part II[2] article.

To this day I keep getting questions on Lisp for the Web. Part of the reason is that I left the original code to rot. I moved on to pursuit other adventures and never really looked back. The result is that the code is now in a sad state where it isn't compatible with newer versions of Hunchentoot. That's not where I wanted it to end. I think the Common Lisp community deserves up-to-date material; I don't want anyone come to my tutorial with the intentions of learning Lisp and leave out of frustration as the examples don't compile.

[1]http://landoflisp.com

[2]http://msnyder.info/posts/2011/07/lisp-for-the-web-part-ii/

What's new?

This book is my attempt to restore Lisp for the Web to its former glory. Besides bringing the code up-to-date, I've also taken this opportunity to switch to HTML5 and include some new material. This additional material includes:

- coverage of interacting with the popular NoSQL MongoDB[3],
- a treatment of the MapReduce[4] algorithm in Lisp and how we can execute it on a remote node without ever leaving our Lisp environment, and
- expanded coverage on how to express JavaScript in Lisp and generate it dynamically, and
- finally, I dive deeper into the Common Lisp Object System (CLOS) and show how it allows us to model complex functionality with simple constructs. The example I include is a variant of the design pattern Observer, but in just one line of standard CLOS code rather than building elaborate class hierarchies.

In the end I hope you enjoy this book and find learning Lisp just as rewarding as I did.

Source code to Lisp for the Web

All source code is available in a Github repository[5]. The code comes in three versions: one with the prototypical backend we're about to develop in the first part of the book, another version that extends the code to use a persistent storage, and finally a module that includes the MapReduce implementation.

If you clone my Github repository you'll notice that I've duplicated several definitions in the different modules. This isn't something I necessarily encourage in production code. Rather my intent is to present an easily digestible, stand-alone snapshot of each version of the application. In the tutorial I take an iterative approach where I grow a live system organically. Often, this includes replacing existing functions with new versions. As such, the duplications are traces of what once was.

Preparations

Get a Lisp

I've organized this book as a tutorial. If you want to follow along you need a Lisp. There are several high-quality alternatives available to you, both open-source as well as commercial implementations. A good starting point is the Getting Started page on the CLiki[6].

[3]http://www.mongodb.org
[4]http://en.wikipedia.org/wiki/MapReduce
[5]https://github.com/adamtornhill/LispForTheWeb
[6]http://www.cliki.net/Getting%20Started

Installing libraries with Quicklisp

Common Lisp has tons of interesting open-source libraries. We'll get to meet some of them in this book. Since the advent of Quicklisp[7], installing a library is a breeze. You can get a copy of Quicklisp here: http://www.quicklisp.org[8]. Once you've downloaded your copy, just follow the instructions on the Cliki[9] to install Quicklisp.

About Adam and Adam

The original Lisp for the Web article was published by an Adam Petersen. Now it may look as though I've stolen not only Mr Petersen's idea but also the bulk of his writings. Trust me, it's OK - I used to be him. I changed my family name in the summer of 2013 as I married my beautiful Jenny.

About Adam Tornhill

Adam is a programmer that combines degrees in engineering and psychology. He's the author of Your Code as a Crime Scene[10], has written the popular Lisp for the Web tutorial and self-published a book on Patterns in C[11]. Adam also writes open-source software in a variety of programming languages. His other interests include modern history, music and martial arts.

Credits

Thanks to Christopher Wellons, Stuart Malcolm and Matthew Malisz for reporting issues in earlier versions of this book.

My amazing wife Jenny Tornhill[12] designed the cover of this book. Thanks Jenny - you rock!

[7]http://www.quicklisp.org
[8]http://www.quicklisp.org
[9]http://www.cliki.net/Getting%20Started
[10]https://pragprog.com/book/atcrime/your-code-as-a-crime-scene
[11]https://leanpub.com/patternsinc
[12]http://www.jennytornhill.se

Lisp for the Web

With his essay Beating the Averages[13], Paul Graham told the story of how his web start-up Viaweb outperformed its competitors by using Lisp. Lisp? Did I parse that correctly? That ancient language with all those scary parentheses? Yes, indeed! And with the goal of identifying its strengths and what they can do for us, I'll put Lisp to work developing a web application. In the process we'll find out how a 50 years old language can be so well-suited for modern web development and yes, it's related to all those parentheses.

What to expect

Starting from scratch, we'll develop a three-tier web application. I'll show how to:

- utilize powerful open source libraries for expressing dynamic HTML5 and JavaScript in Lisp,
- develop a small, embedded domain specific language tailored for my application,
- extend the typical development cycle by modifying code in a running system and execute code during compilation,
- migrate from data structures in memory to persistent objects using a third party NoSQL database (MongoDB), and
- finally show how we can execute a MapReduce algorithm on a remote database server without even leaving our Lisp environment.

I'll do this in a live system transparent to the users of the application. Because Lisp is so high-level, I'll be able to achieve everything in just around 80 lines of code.

This article will not teach you Common Lisp (for that purpose I provide some book recommendations at the end). Instead, I'll give a short overview of the language and try to explain the concepts as I introduce them, just enough to follow the code. The idea is to convey a feeling of how it is to develop in Lisp rather than focusing on the details.

The Lisp story

Lisp is actually a family of languages discovered by John McCarthy over 50 years ago. The characteristic of Lisp is that Lisp code is made out of Lisp data structures with the practical implication that it is not only natural, but also highly effective to write programs that write programs. This feature has allowed Lisp to adapt over the years. For example, as object-oriented programming became popular, powerful object systems could be implemented in Lisp as libraries

[13] http://www.paulgraham.com/avg.html

without any change to the core language. Later, the same proved to be true for aspect-oriented programming.

This idea is not only applicable to whole paradigms of programming. Its true strength lays in solving everyday problems. With Lisp, it's straightforward to build-up a domain specific language allowing us to program as close to the problem domain as our imagination allows. I'll illustrate the concept soon, but before we kick-off, let's look closer at the syntax of Lisp.

Crash course in Lisp

What Graham used for Viaweb was Common Lisp, an ANSI standardized language, which we'll use in this article too (the other main contenders are Scheme, generally considered cleaner and more elegant but with a much smaller library, and Clojure that introduces several interesting concepts and targets existing VMs).

Common Lisp is a high-level interactive language that may be either interpreted or compiled. You interact with Lisp through its top-level . The top-level is basically a prompt. On my system it looks like this:

```
CL-USER>
```

Through the top-level, we can enter expressions and see the results (the values returned by the top-level are highlighted):

```
CL-USER>(+ 1 2 3)
6
```

As we see in the example, Lisp uses a prefix notation. A parenthesized expression is referred to as a form . When fed a form such as *(+ 1 2 3)* , Lisp generally treats the first element (+) as a function and the rest as arguments. The arguments are evaluated from left to right and may themselves be function calls:

```
CL-USER>(+ 1 2 (/ 6 2))
6
```

We can define our own functions with *defun*:

```
CL-USER>(defun say-hello (to)
          (format t "Hello, ~a" to))
```

Here we're defining a function *say-hello* , taking one argument: *to* . The *format* function is used to print a greeting and resembles a *printf* on steroids. Its first argument is the output stream and here we're using *t* as a shorthand for standard output. The second argument is a string, which in our case contains an embedded directive ∼*a* instructing format to consume one argument and output it in human-readable form. We can call our function like this:

```
CL-USER>(say-hello "ACCU")
Hello, ACCU
NIL
```

The first line is the side-effect, printing *"Hello, ACCU"* and *NIL* is the return value from our function. By default, Common Lisp returns the value of the last expression. From here we can redefine say-hello to return its greeting instead:

```
CL-USER>(defun say-hello (to)
          (format nil "Hello, ~a" to))
```

With *nil* as its destination, format simply returns its resulting string:

```
CL-USER>(say-hello "ACCU")
"Hello, ACCU"
```

Now we've gotten rid of the side-effect. Programming without side-effects is in the vein of functional programming, one of the paradigms supported by Lisp. Lisp is also dynamically typed. Thus, we can feed our function a number instead:

```
CL-USER>(say-hello 42)
"Hello, 42"
```

In Lisp, functions are first-class citizens. That means, we can create them just like any other object and we can pass them as arguments to other functions. Such functions taking functions as arguments are called higher-order functions . One example is *mapcar* . *mapcar* takes a function as its first argument and applies it subsequently to the elements of one or more given lists:

```
CL-USER>(mapcar #'say-hello (list "ACCU" 42 "Adam"))
("Hello, ACCU" "Hello, 42" "Hello, Adam")
```

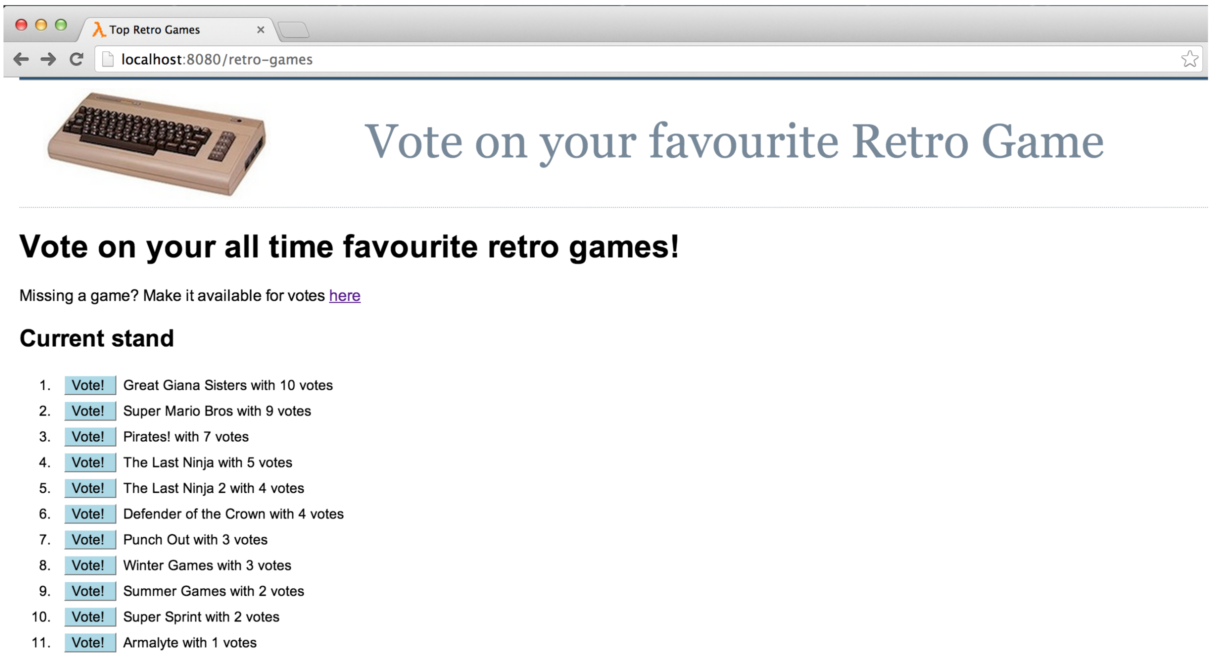
The funny #' is just a shortcut for getting at the function object. As you see above, *mapcar* collects the result of each function call into a list, which is its return value. This return value may of course serve as argument to yet another function:

```
CL-USER>(sort (mapcar #'say-hello (list "ACCU" 42 "Adam")) #'string-lessp)
("Hello, 42" "Hello, ACCU" "Hello, Adam")
```

Lisp itself isn't hard, although it may take some time to wrap ones mindset around the functional style of programming. As you see, Lisp expressions are best read inside-out. But the real secret to understanding Lisp syntax is to realize that the language doesn't have one; what we've been entering above is basically parse-trees, generated by compilers in other languages. And, as we'll see soon, exactly this feature makes it suitable for metaprogramming.

The Brothers are History

Remember the hot gaming discussions 20 years ago? “Giana Sisters” really was way better than “Super Mario Bros”, wasn’t it? We’ll delegate the question to the wise crowd by developing a web application. Our web application will allow users to add and vote for their favourite retro games. A screenshot of the end result is provided in Figure 1 below.



Vote on your all time favourite retro games!

Missing a game? Make it available for votes here

Current stand

1. Vote! Great Giana Sisters with 10 votes
2. Vote! Super Mario Bros with 9 votes
3. Vote! Pirates! with 7 votes
4. Vote! The Last Ninja with 5 votes
5. Vote! The Last Ninja 2 with 4 votes
6. Vote! Defender of the Crown with 4 votes
7. Vote! Punch Out with 3 votes
8. Vote! Winter Games with 3 votes
9. Vote! Summer Games with 2 votes
10. Vote! Super Sprint with 2 votes
11. Vote! Armalyte with 1 votes

The Retro Games front page

From now on, I start to persist my Lisp code in textfiles instead of just entering expressions into the top-level. Further, I define a package for my code. Packages are similar to namespaces in C++ or Java’s packages and helps to prevent name collisions (the main distinction is that packages in Common Lisp are first-class objects).

```
(defpackage :retro-games
  (:use :cl :cl-who :hunchentoot :parenscript))
```

The new package is named *:retro-games* and I also specify other packages that we’ll use initially:

- CL is Common Lisp’s standard package containing the whole language.
- CL-WHO[14] is a library for converting Lisp expressions into XHTML.
- Hunchentoot[15] is a web-server, written in Common Lisp itself, and provides a toolkit for building dynamic web sites.
- Parenscript[16] allows us to compile Lisp expressions into JavaScript. We’ll use this for client-side validation.

[14]http://weitz.de/cl-who/
[15]http://weitz.de/hunchentoot/
[16]http://common-lisp.net/project/parenscript/

Using Quicklisp, installing these libraries is as simple as it gets:

```
CL-USER> (ql:quickload '(cl-who hunchentoot parenscript))
```

The *quickload* function will ensure that the listed libraries are properly installed, transparently resolving and installing their dependencies.

With my package definition in place, I'll put the rest of the code inside it by switching to the *:retro-games* package:

```
(in-package :retro-games)
```

Most top levels indicate the current package in their prompt. On my system the prompt now looks like this:

```
RETRO-GAMES>
```

Representing Games

With the package in place, we can return to the problem. It seems to require some representation of a game and I'll choose to abstract it as a class:

```
(defclass game ()
  ((name  :initarg  :name)
   (votes :initform 0)))
```

The expression above defines the class game without any user-specified superclasses, hence the empty list *()* as second argument. A game has two slots (slots are similar to attributes or members in other languages): a *name* and the number of accumulated *votes* . To create a game object I invoke *make-instance* and passes it the name of the class to instantiate:

```
RETRO-GAMES>(defvar many-lost-hours (make-instance 'game :name "Tetris"))
MANY-LOST-HOURS
```

Because I specified an initial argument in my definition of the name slot, I can pass this argument directly and initialize that slot to *"Tetris"*. The votes slot doesn't have an initial argument. Instead I specify the code I want to run during instantiation to compute its initial value through *:initform*. In this case the code is trivial, as I only want to initialize the number of votes to zero. Further, I use *defvar* to assign the object created by *make-instance* to the variable *many-lost-hours*.

Now that we got an instance of game we would like to do something with it. We could of course write code ourselves to access the slots. However, there's a more lispy way; *defclass* provides the possibility to automatically generate accessor functions for our slots:

```
(defclass game ()
  ((name  :reader   name
          :initarg  :name)
   (votes :accessor votes
          :initform 0)))
```

The option *:reader* in the name slot will automatically create a read function and the option *:accessor* used for the votes slot will create both read and write functions. Lisp is pleasantly uniform in its syntax and these generated functions are invoked just like any other function:

```
RETRO-GAMES>(name many-lost-hours)
"Tetris"
RETRO-GAMES>(votes many-lost-hours)
0
RETRO-GAMES>(incf (votes many-lost-hours))
1
RETRO-GAMES>(votes many-lost-hours)
1
```

The only new function here is *incf* , which when given one argument increases its value by one. We can encapsulate this mechanism in a method used to vote for the given game:

```
(defmethod vote-for (user-selected-game)
  (incf (votes user-selected-game)))
```

The top-level allows us to immediately try it out and vote for Tetris:

```
RETRO-GAMES>(votes many-lost-hours)
1
RETRO-GAMES>(vote-for many-lost-hours)
2
RETRO-GAMES>(votes many-lost-hours)
2
```

A prototypic backend

Before we can jump into the joys of generating web pages, we need a backend for our application. Because Lisp makes it so easy to modify existing applications, it's common to start out really simple and let the design evolve as we learn more about the problem we're trying to solve. Thus, I'll start by using a list in memory as a simple, non-persistent storage.

```
(defvar *games* '())
```

The expression above defines and initializes the global variable (actually the Lisp term is special variable) **games** to an empty list. The asterisks aren't part of the syntax; it's just a naming convention for globals. Lists may not be the most efficient data structure for all problems, but Common Lisp has great support for lists and they are easy to work with. Later we'll change to a real database and, with that in mind, I encapsulate the access to **games** in some small functions:

```
(defun game-from-name (name)
  (find name *games* :test #'string-equal
                :key  #'name))
```

Our first function *game-from-name* is implemented in terms of *find*. *find* takes an item and a sequence. Because we're comparing strings I tell find to use the function *string-equal* for comparison (remember, #' is a short cut to refer to a function). I also specify the key to compare. In this case, we're interested in comparing the value returned by the *name* function on each game object.

If there's no match *find* returns *NIL*, which evaluates to *false* in a boolean context. That means we can reuse *game-from-name* when we want to know if a game is stored in the **games** list. However, we want to be clear with our intent:

```
(defun game-stored? (game-name)
  (game-from-name game-name))
```

As illustrated in Figure 1, we want to present the games sorted on popularity. Using Common Lisp's *sort* function this is pretty straightforward; we only have to take care, because for efficiency reasons *sort* is destructive. That is, *sort* is allowed to modify its argument. We can preserve our **games** list by passing a copy to *sort*. I tell *sort* to return a list sorted in descending order based on the value returned by the *votes* function invoked on each game:

```
(defun games ()
  (sort (copy-list *games*) #'> :key #'votes))
```

So far the queries. Let's define one more utility for actually adding games to our storage:

```
(defun add-game (name)
  (unless (game-stored? name)
    (push (make-instance 'game :name name) *games*)))
```

push is a modifying operation and it prepends the game instantiated by *make-instance* to the **games** list. Let's try it all out at the top level.

```
RETRO-GAMES>(games)
NIL
RETRO-GAMES>(add-game "Tetris")
(#<GAME @ #x71b943c2>)
RETRO-GAMES>(game-from-name "Tetris")
#<GAME @ #x71b943c2>
RETRO-GAMES>(add-game "Tetris")
NIL
RETRO-GAMES>(games)
(#<GAME @ #x71b943c2>)
RETRO-GAMES>(mapcar #'name (games))
("Tetris")
```

Customizing the printed representation of CLOS objects

The values returned to the top level may not look too informative. It's basically the printed representation of a game object. Common Lisp allows us to customize how an object shall be printed. That's done by specializing the generic function *print-object* for our *game* class:

```
(defmethod print-object ((object game) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (name votes) object
      (format stream "name: ~s with ~d votes" name votes))))
```

With this specialisation in place, our game objects now look more informative in the REPL:

```
#<GAME name: "Tetris" with 2 votes>
```

The *print-object* specialisation introduces a few utilities from the standard library:

- *print-unreadable-object* helps us with the output stream set-up and displays type information in the printout.
- *with-slots* lets us reference the slots in the *game* instance as though they were variables.

Particularly *with-slots* is an example of how Lisp is grown to avoid all signs of duplication, no matter how subtle they may be (without *with-slots* we would have to access the *game* object twice in *format*).

We'll soon learn how to grow our own extensions to Lisp. But first, with this prototypic backend in place, we're prepared to enter the web.