LISP

A Programmable Programming Language

Christoph Müller

25.01.2017

Offenes Kolloquium für Informatik

Agenda

1. Why Lisp?
2. The Basics of Lisp
3. Macros in Action
4. Tools and Platforms
5. Literature and more obscure Lisp dialects
6. Why it never (really) caught on
7. Bonus - A bit of History

Why Lisp?

Timeless

- When we talk about Lisp we talk about a language family
- One of the oldest (∼ 1958) language families still in use today (only Fortran is older)
- The Syntax is by its very nature timeless

Timeline of Lisp dialects (edit)

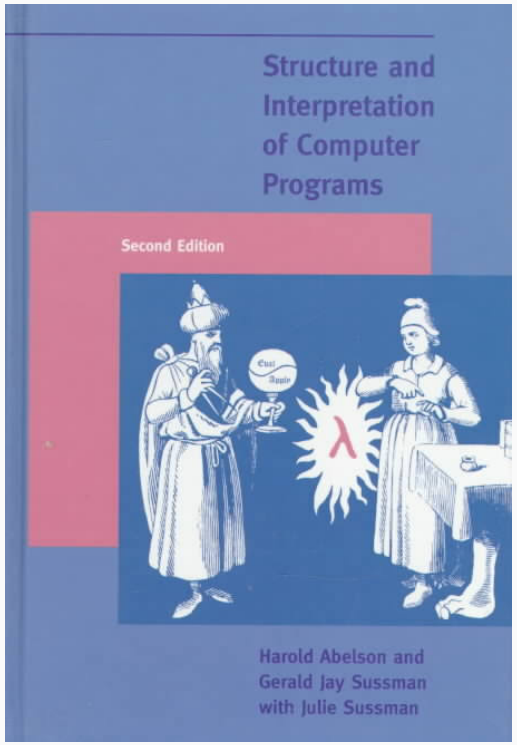
	1955	1960	1965	1970	1975	1980	1985	1990	1995	2000	2005	2010	2015
Lisp 1.5	Lisp 1.5												
Maclisp			Maclisp										
Interlisp				Interlisp									
ZetaLisp					Lisp Machine Lisp								
Scheme					Scheme								
NIL					NIL								
Common Lisp						Common Lisp							
T							T						
Emacs Lisp							Emacs Lisp						
AutoLISP							AutoLISP						
ISLISP							ISLISP						
EuLisp							EuLisp						
PicoLisp								PicoLisp					
Racket									Racket				
Arc										Arc			
Clojure											Clojure		
LFE											LFE		
Hy												Hy	

Innovator

- Garbage Collection
- Homoiconicity (Code is Data)
- Higher Order Functions
- Dynamic Typing
- Read Evaluate Print Loop (REPL)
- Multiple Dispatch
- And many more ...

Scheme - A Language for Teaching

- Scheme was used as an introductory Language in famous Universities like MIT (6.001)
- Extremely small language core
- Great for learning to build your own abstractions



Picking a Language for this Talk

Lets look at the most popular Lisp dialects on GitHub (provided by GitHut):

GitHub Popuplarity Rank	Language
20	Emacs Lisp
23	**Clojure**
40	**Scheme**
42	Common Lisp
48	Racket

Clojure with its JVM heritage and Scheme with its focus on a small core will be used throughout this talk.

The Basics of Lisp

The name giving lists

- The basis of lisp is the s(ymbolic)-expression
- Either a atom or a list
- Atoms are either symbols or literals
- Every element of a list is either an atom or another list
- Elements are separated by whitespace
- The first element of a (to be evaluated) list has to be what we will call a *verb* in this talk

```
;atoms
x
12
;lists
(+ 1 2 3)
(+ (* 2 3) 3)
```

What is a verb?

- A verb is either a
  - A function
  - A macro
  - A special form
- Special forms include *if*, *fn*, *loop*, *recur etc.*
- They are built into the language and cannot be user defined
- On the other hand functions and macros can be
- Since functions are familiar to most people we will start with them

Calling Functions

- The arguments of functions are evaluated before they are passed to the function
- This is an important distinction from macros/special forms
- Calling functions in a prefix manner might feel strange in the beginning

```
;the + function called as a
;prefix and not as an infix
(+ 1 2 3)
;the infamous
(println "hello world")
```

Calling Java Methods – Clojure Only

- Since Clojure runs on the JVM, interop with Java is necessary to make use of existing libraries
- Java Methods are called like (.instanceMember instance args*)

```
(.toUpperCase "Hello World")
-> "HELLO WORLD"
```

- Creating a new Instance will be very familiar to Java Developers
- There is however a short form for creating new instances

```
(new String "hello world")
-> "hello world"
(String. "hello world")
-> "hello world"
```

Just a bit more Syntax

- Before we will learn how to create our own functions a bit more syntastic sugar
- Vectors are the data structure in Clojure that are used to define the arguments of a function

```
[1 2 3]
-> [1 2 3]
(vector 1 2 3)
-> [1 2 3]
```

- Maps/Dictionaries are created via the curly brace literal

```
{"a" 1 "b" 2 "c" 3}; or (hash-map ...)
-> {"a" 1, "b" 2, "c" 3}
; note the comma, comma is whitespace in Clojure
```

- These are implemented via so called reader macros we will learn about them in the macro section

Define your own Functions - 1

- The special form *fn* is used to create functions

```
(fn [x] (* x x))
-> #function[user/eval10725/fn--10726]
((fn [x] (* x x)) 12)
-> 144
```

- An optional name can be given to the function to make non tail calls

```
((fn foo [x] (if (< x 1) x (foo (dec x)))) 10)
-> 0
```

Define your own Functions - 2

- to make a tail recursive call the *recur* special form is used

```
((fn [x] (if (< x 1) x (recur (dec x)))) 10)
```

- Since functions will often be bound to a global variable (inside a namespace) the following syntax will be seen often

```
(defn foo "doc string here" [x]
  (if (< x 1)
    x
    (foo (dec x))))
-> #'user/foo
(foo 10)
-> 0
```

Define your own Functions - 3

- For short lambda functions there is an even more compact notation
- inside the lambda function % is used to for arguments
- % and %1 are used for the first argument, %2 ... for the rest

```
#(* % %)
-> #function[user/eval10725/fn--10726]
(map #(* % %) (range 10))
-> (0 1 4 9 16 25 36 49 64 81)
```

Branch with *if*

- We have already seen the *if* special form
- It consists of a test, a then expression and an optional else expression
- *if* can be used like a ternary expression in Java

```
(println (if (< 4 3) "hello" "world"))
-> world
```

```
System.out.println(4 < 3 ? "hello" : "world")
```

do multiple things

- Evaluates multiple expressions and returns the value of the last one (or nil)

```
(if (< 3 4)
    (do
        (println "hello world")
        (println "and again")))
-> hello world
   and again
```

Bind with let

- Of course we also need to bind local variables inside expressions
- The *let* special form is used for that
- It uses pairs inside a vector for that purpose
- Has support for Destructering

```
(let [x 1] x)
-> 1
;basic Destructering
(let [[x y] [1 2]] (+ x y))
-> 3
```

Loop with … well … loop

- We have seen recursion, now we will cover iteration with the *loop* special form
- The *loop* form is very similiar to a *let* binding
- To repeat we use *recur* just like when working with tail recursion earlier

```
(loop [x 10]
  (if (> x 1)
    (recur (- x 2))))
```

- There are other types of loops in clojure, like *for* and *while*, but they are implemented as macros
- *loop* and *recur* is therefore all we need!

Your new best friends *doc* and *source*

- *doc* will show you the docstring of a given function, macro or special form

```
(doc +)
-> ([] [x] [x y] [x y & more])
   Returns the sum of nums. (+) returns 0.
   Does not auto-promote longs,
   will throw on overflow. See also +'
```

- *source* will show you the source code of a given function or macro

```
(source when)
-> (defmacro when
     ".. doc string ..."
     [test & body]
     (list 'if test (cons 'do body)))
```

Macros in Action

Kinds of Macros

Macros can be grouped in different Categories

- Syntactic Sugar Macros - Using simple pattern matching and templates
- Complex Transformations - The most demanding and the most rewarding
- Reader Macros - Syntactic sugar on the reader level, not to be confused with the other two

Yes Code really is Data

- Code really is nothing more than a linked list

```
(type '(+ 1 2 3))
-> clojure.lang.PersistentList
```

- The ’ is used to **prevent** evaluation, it is equivalent to *(quote ...)*
- The function *eval* (may be familiar from a lot of scripting languages) takes a s-expression, not a string!

```
(eval '(+ 1 2 3))
-> 6
```

Reader Macros

- To get s-expression from a string, the *read-string* function can be used

```
(eval (read-string "(+ 1 2 3)"))
-> 6
```

- The reader uses read macros to parse special syntax like [], the ’ or the lambda #() syntax
- Clojure has a set of predefined reader macros, they can not be user defined
- Some lisps (e.g. Common Lisp or Racket) don’t suffer from this restriction
- That means that theses lisp dialects have compile time + parse time macros

The syntax quote

- Before we will look at macros, we will introduce the syntax-quote
- It helps us evaluating things in nested quoted structeres

```
; ` is a syntax quote
; ~ evaluates inside such quote
`(foo ~(+ 1 2 3))
-> (user/foo 6)
```

- Everything that is not evaluated through the tilde character will be left alone (only the current namespace is added)

Macros

- Macros are arbitrary lisp code executed at compile time
- Normally only code transformation is done, but it is not limited to transformations
- One could for instance query a database or perform computation of all sorts

```
(defmacro foo [] (+ 1 2))
-> 3
```

A bit of sugar

- Let's look at the simple *when* macro with *(source when)*

```
(defmacro when
  ...
  [test & body]
  (list 'if test (cons 'do body)))
```

- The ampersand just stands for the rest, so body are all expressions after test.
- Here we create code by creating a list
- Using the list function is basically the inverse of a syntax quote, everything is evaluated except quoted expressions

The *while* loop

- The *while* loop ships with Clojure, here is the source

```
(defmacro while
  ...
  [test & body]
  `(loop []
     (when ~test
       ~@body
       (recur))))
```

- The loop does not need bindings, since were ware dealing with a while and not a for loop
- The @-sign in front of body unpacks the body list into its elements
- So 1 2 3 4 ...instead of (1 2 3 4 ...)

Pattern Matching in Scheme

- Scheme provides an even more elegant syntax for simple macros with *syntax-rules*

```
(define-syntax for
  (syntax-rules (in as)
      (
       ;pattern
       (for element in list body ...)
       ;template
       (map (lambda (element) body ...) list)
      )
      ((for list as element body ...)
       (for element in list body ...))))

(for i in '(1 2 3) (display i))
```

Complex Macros

- For time reasons we can’t look at more complex macros in detail
- We will look at an example at a higher level
- As an example I have picked the async/await statement from languages like C# or JS (ECMAScript 2017 draft)
- This is usally done by creating a state machine
- The CSharp code was decompiled to retrieve the state machine

Async in C# - Before

```
static async Task<int> TestAsync()
{
    Console.WriteLine("Init test method");

    var firstResult = await GetNumberAsync(1);
    Console.WriteLine(firstResult);

    var secondResult = await GetNumberAsync(2);
    Console.WriteLine(secondResult + firstResult);

    var thirdResult = await GetNumberAsync(4);

    Console.WriteLine("I'm done");
    return firstResult - secondResult + thirdResult;
}

public static async Task<int> GetNumberAsync(int number)
    => await Task.Run(() => number);
```

Async in C# - After

The CLR has no extra byte code instructions for async/await, everything is handled by the compiler

```
[CompilerGenerated] // shortend
void IAsyncStateMachine.MoveNext()
{
    try
    {
        switch (num)
        {
        case 0:
            taskAwaiter = this.<>u__1;
            this.<>u__1 = default(TaskAwaiter<int>);
            this.<>1__state = -1;
            break;
        case 1:
            taskAwaiter2 = this.<>u__1;
            this.<>u__1 = default(TaskAwaiter<int>);
            this.<>1__state = -1;
            goto IL_117;
```

Async in Clojure with core.async

- In Clojure we don't need dont to wait until a standard comittee adds the feature
- We just use the core.async library implemented purely in Clojure
- core.async uses channels in a technique known as Communicating Sequential Processes
- The syntax will be more familiar to users of the go language

```clojure
(defn what-is-the-answer [c]
  (go
    ;timeout creates a new channel
    (<! (timeout 2.3652E17))
    (>! c 42)))
```

The *macroexpand* function helps us to examine the macro code

```
(fn state-machine
  ([state_3730]
    (loop []
      (let
        [result
          (case (int (ioc/aget-object state_3730 1))
           3 (let [inst_3728 (ioc/aget-object state_3730 2)
                   state_3730 state_3730]
               (ioc/return-chan state_3730 inst_3728))
           2 (let [inst_3725 (ioc/aget-object state_3730 2)
                   inst_3726 (vector kind query)
                   state_3730 (ioc/aset-all! state_3730 5 inst_3725)]
               (ioc/put! state_3730 3 c inst_3726))
           1 (let [inst_3722 (rand-int 100)
                   inst_3723 (timeout inst_3722)
                   state_3730 state_3730]
               (ioc/take! state_3730 2 inst_3723)))]
```

Deep Understanding

Clojure provides a deep understanding of the language through macros and functions like *doc*, *source* and *macroexpand*. This should not be taken for granted, especially when compared to languages like e.g. C++.

C++ - The worst Offender

> *Help me sort out the meaning of "{}" as a constructor argument*
>
> – Scott Meyers, Author of *Effective C++*

```
class Widget{
    public:
        // default ctor
        Widget();
        // std::initializer_list ctor
        Widget(std::initializer_list<int> il);
};

Widget w1;      // calls default ctor
Widget w2{};    // also calls default ctor
Widget w3();    // most vexing parse! declares a function!

Widget w4({});  // calls std::initializer_list ctor with empty list
Widget w5{{}};  // ditto  -- ... not so fast Dr. Meyers
```

C++ - The worst Offender

- The specific example can be looked up on Scott Meyers Blog
- The last call does *not* create an empty list
- Even a seasoned C++ expert and book author can't figure out seemingly simple examples
- -> Please don't take the tools Clojure provides for granted

Be carefull with Macros

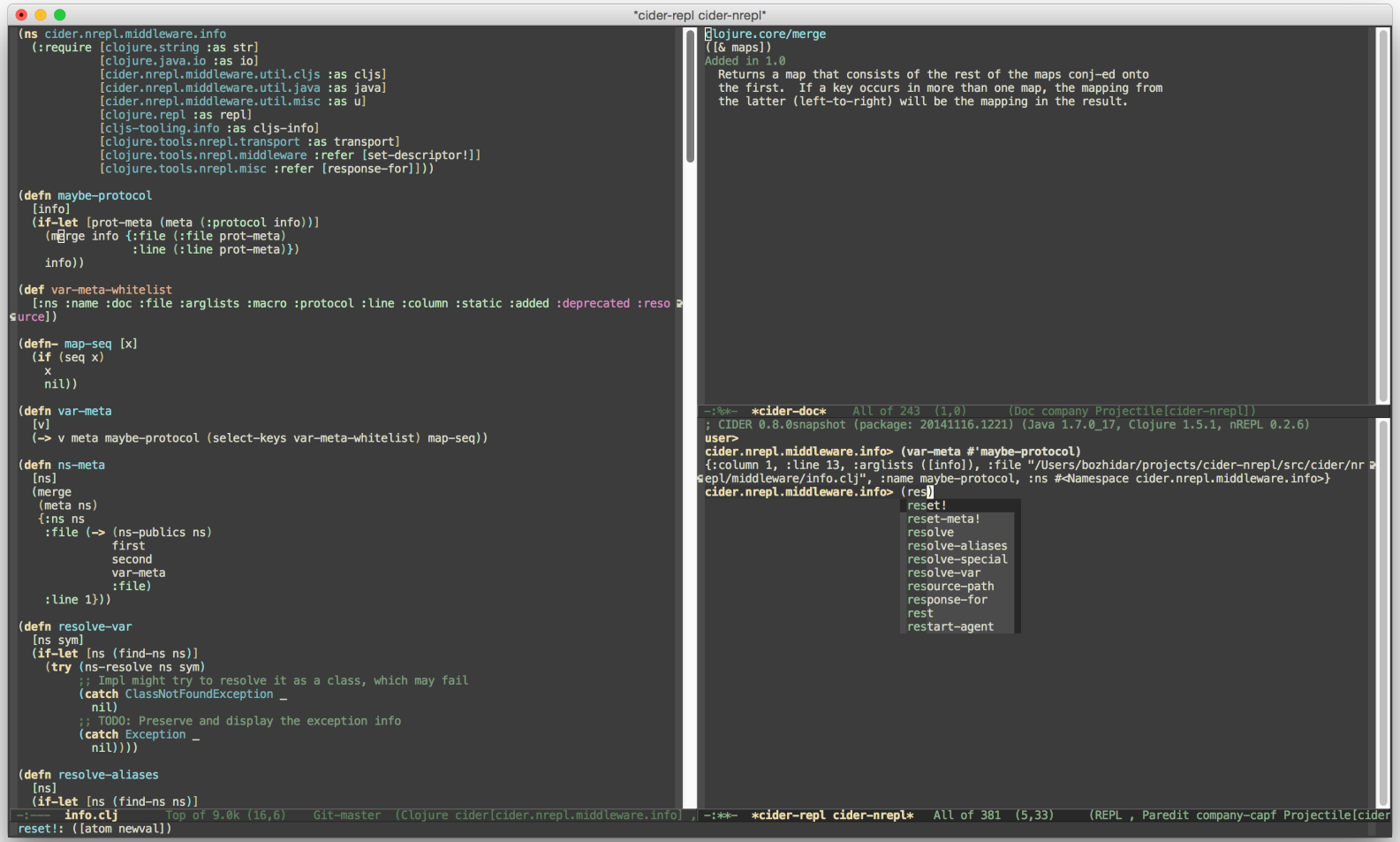
- Use them only when a function won’t do
- Macros tend to “creep” up the call chain
- Writing good Macros takes quite a bit of practice
- Good Error messages (hello Rust) are now your responsibility
- Since you are now the “compiler engineer”

Tools and Platforms

Emacs

- The most important IDE for Lisp is of course emacs
- Written in EmacsLisp and with support for pretty much every Lisp dialect
- For Clojure the cider package is recommended

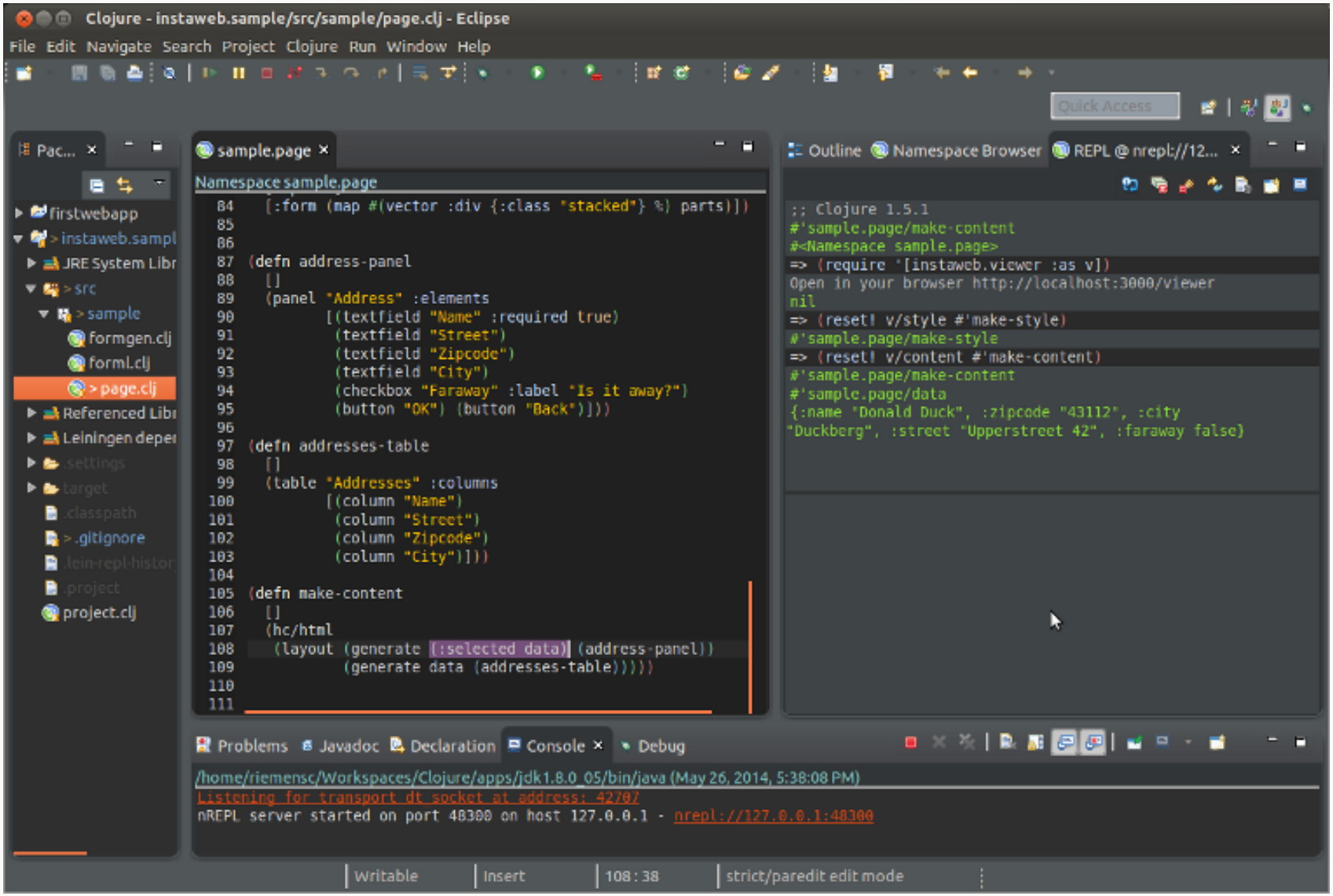
Emacs


cider-repl cider-nrepl
(ns cider.nrepl.middleware.info
(:require [clojure.string :as str]
[clojure.java.io :as io]
[cider.nrepl.middleware.util.cljs :as cljs]
[cider.nrepl.middleware.util.java :as java]
[cider.nrepl.middleware.util.misc :as u]
[clojure.repl :as repl]
[cljs-tooling.info :as cljs-info]
[clojure.tools.nrepl.transport :as transport]
[clojure.tools.nrepl.middleware :refer [set-descriptor!]]
[clojure.tools.nrepl.misc :refer [response-for]]))
(defn maybe-protocol
[info]
(if-let [prot-meta (meta (:protocol info))]
(merge info {:file (:file prot-meta)
:line (:line prot-meta)})
info))
(def var-meta-whitelist
[:ns :name :doc :file :arglists :macro :protocol :line :column :static :added :deprecated :resource])
(defn- map-seq [x]
(if (seq x)
x
nil))
(defn var-meta
[v]
(-> v meta maybe-protocol (select-keys var-meta-whitelist) map-seq))
(defn ns-meta
[ns]
(merge
(meta ns)
{:ns ns
:file (-> (ns-publics ns)
first
second
var-meta
:file)
:line 1}))
(defn resolve-var
[ns sym]
(if-let [ns (find-ns ns)]
(try (ns-resolve ns sym)
;; Impl might try to resolve it as a class, which may fail
(catch ClassNotFoundException _
nil)
;; TODO: Preserve and display the exception info
(catch Exception _
nil))))
(defn resolve-aliases
[ns]
(if-let [ns (find-ns ns)]
-:---
info.clj
Top of 9.0k (16,6)
Git-master
(Clojure cider[cider.nrepl.middleware.info]
reset!: ([atom newval])
clojure.core/merge
([& maps])
Added in 1.0
Returns a map that consists of the rest of the maps conj-ed onto
the first. If a key occurs in more than one map, the mapping from
the latter (left-to-right) will be the mapping in the result.
-:%*-
cider-doc
All of 243 (1,0)
(Doc company Projectile[cider-nrepl])
; CIDER 0.8.0snapshot (package: 20141116.1221) (Java 1.7.0_17, Clojure 1.5.1, nREPL 0.2.6)
user>
cider.nrepl.middleware.info> (var-meta #'maybe-protocol)
{:column 1, :line 13, :arglists ([info]), :file "/Users/bozhidar/projects/cider-nrepl/src/cider/nrepl/middleware/info.clj", :name maybe-protocol, :ns #<Namespace cider.nrepl.middleware.info>}
cider.nrepl.middleware.info> (res
reset!
reset-meta!
resolve
resolve-aliases
resolve-special
resolve-var
resource-path
response-for
rest
restart-agent
-:**-
cider-repl cider-nrepl
All of 381 (5,33)
(REPL , Paredit company-capf Projectile[cider

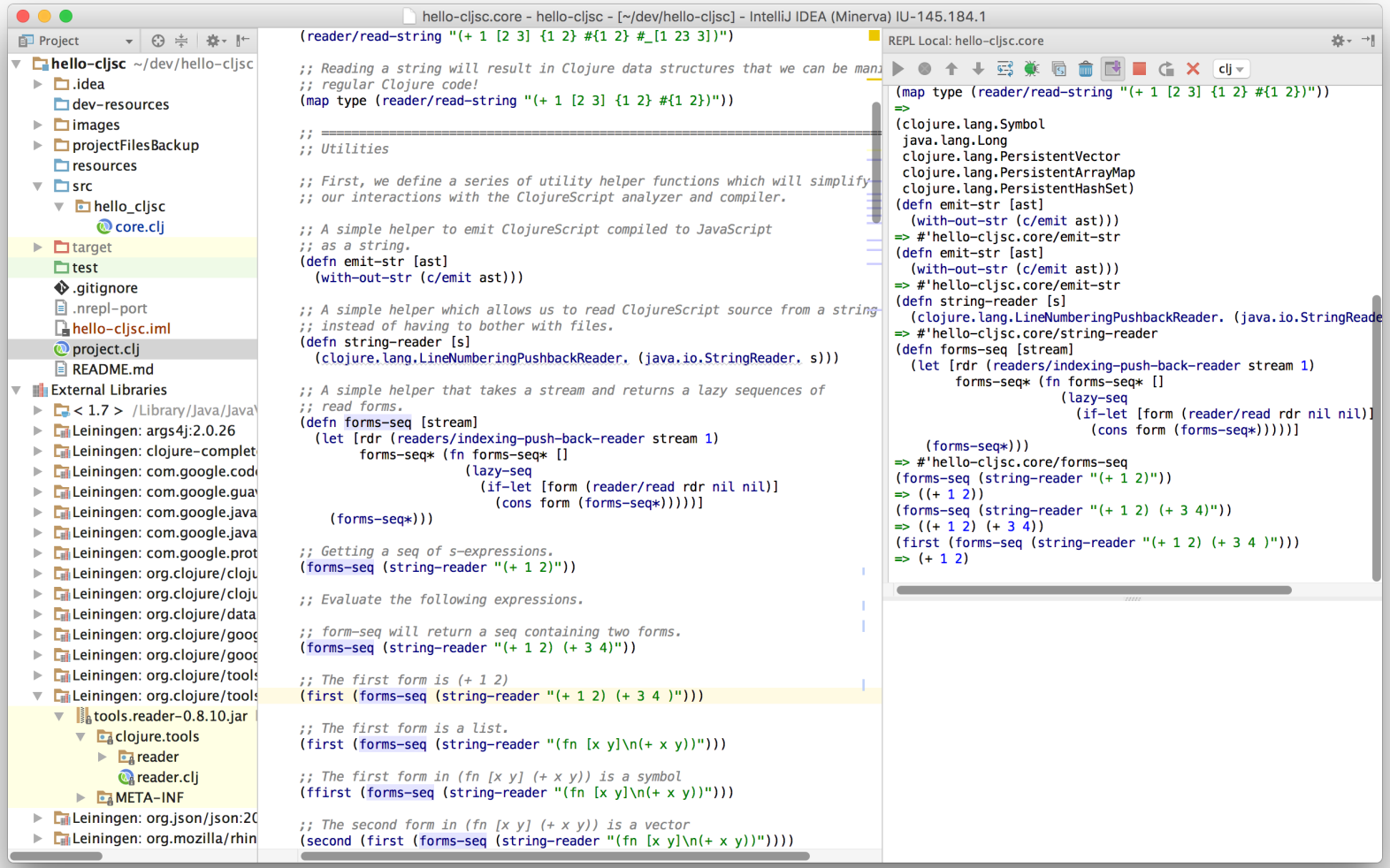
Clojure IDE Support

- For Clojure the two famous IDEs Eclipse and IntelliJ both have support via plugins
- Eclipse has the Counter Clockwise Wise Plugin
- IntelliJ has the (commercial) Cursive IDE

Eclipse


Clojure - instaweb.sample/src/sample/page.clj - Eclipse
File Edit Navigate Search Project Clojure Run Window Help
Quick Access
Pac...
firstwebapp
instaweb.sampl
JRE System Libr
src
sample
formgen.clj
forml.clj
page.clj
Referenced Libr
Leiningen depe
.settings
target
.classpath
.gitignore
.lein-repl-histor
.project
project.clj
sample.page
Namespace sample.page
84 [:form (map #(vector :div {:class "stacked"} %) parts)])
85
86
87 (defn address-panel
88 []
89 (panel "Address" :elements
90 [(textfield "Name" :required true)
91 (textfield "Street")
92 (textfield "Zipcode")
93 (textfield "City")
94 (checkbox "Faraway" :label "Is it away?")
95 (button "OK") (button "Back")]))
96
97 (defn addresses-table
98 []
99 (table "Addresses" :columns
100 [(column "Name")
101 (column "Street")
102 (column "Zipcode")
103 (column "City")]))
104
105 (defn make-content
106 []
107 (hc/html
108 (layout (generate [:selected data]) (address-panel))
109 (generate data (addresses-table)))))
110
111
Outline
Namespace Browser
REPL @ nrepl://12...
;; Clojure 1.5.1
#'sample.page/make-content
#<Namespace sample.page>
=> (require '[instaweb.viewer :as v])
Open in your browser http://localhost:3000/viewer
nil
=> (reset! v/style #'make-style)
#'sample.page/make-style
=> (reset! v/content #'make-content)
#'sample.page/make-content
#'sample.page/data
{:name "Donald Duck", :zipcode "43112", :city
"Duckberg", :street "Upperstreet 42", :faraway false}
Problems
Javadoc
Declaration
Console
Debug
/home/riemensc/Workspaces/Clojure/apps/jdk1.8.0_05/bin/java (May 26, 2014, 5:38:08 PM)
Listening for transport dt_socket at address: 42707
nREPL server started on port 40300 on host 127.0.0.1 - nrepl://127.0.0.1:40300
Writable
Insert
108 : 38
strict/paredit edit mode

IntelliJ


hello-cljsc.core - hello-cljsc - [~/dev/hello-cljsc] - IntelliJ IDEA (Minerva) IU-145.184.1
Project
hello-cljsc ~/dev/hello-cljsc
.idea
dev-resources
images
projectFilesBackup
resources
src
hello_cljsc
core.clj
target
test
.gitignore
.nrepl-port
hello-cljsc.iml
project.clj
README.md
External Libraries
REPL Local: hello-cljsc.core
clj

Vim

- It is of course heresy to use Vim for Lisp, but I do it anyway
- Tim Pope has written the great fireplace.vim plugin
- Will not be considered a full blown IDE by most people

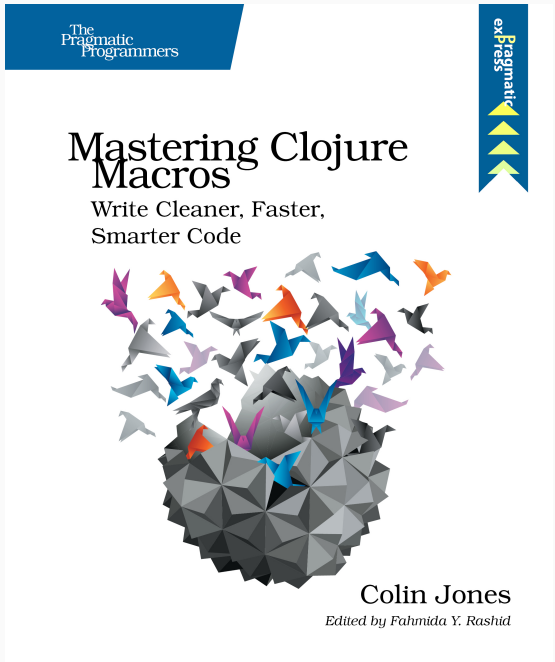
Tools

- Most Lisps have tools for project and package/dependency management
- For Clojure there is leiningen and boot, with leiningen being the older and more popular tool
- The key difference is the declarative approach with leiningen, while boot just uses plain clojure logic
- For the web developers there is an analogy: leiningen -> grunt, boot -> gulp

Literature and more obscure Lisp dialects

Books on Clojure

The left one is available for free from `clojure-buch.de`



Shen Language

- If clojure and scheme are not enough for you, take a look at `shenlanguage.org`
- We will briefly look at a few features
- Static type checking based on the sequent calculus
- Integrated fully functional Prolog (defprolog ...)
- Inbuilt compiler-compiler (Shen-YACC) based on BNF notation
- Can be used to develop efficient compilers for programming languages and transducers for natural language processing
- ...The “Everything but the kitchen sink”-Lisp

Why it never (really) caught on

Performance

> *A Lisp programmer knows the value of everything, but the cost of nothing.*
>
> – Alan Perlis, American Computer Scientist

- Historically a lot of Lisps features required quite a bit of processing power or special hardware support (see Lisp machines)
- Today that argument is mostly moot
- The performance of e.g. Clojure is in the same ballpark as most dynamic JVM languages

Syntax

One of the ideas I keep stressing in the design of Perl is that things that ARE different should LOOK different. The reason many people hate programming in Lisp is because everything looks the same. I've said it before, and I'll say it again: Lisp has all the visual appeal of oatmeal with fingernail clippings mixed in. (Other than that, it's quite a nice language.)

– Larry Wall, Creator of the Perl Language

Syntax

- Maybe a gate keeper for language features (a committee or a “benevolent dictator”) is exactly what you want
- In real businesses it is important that existing code can be understood and altered quickly and cost effectively
- Evolving languages with fixed feature sets may help in the hiring process

Tooling

- A lot of Lisp dialects (especially Scheme implementations) tend to isolate themselves from other eco systems
- Most people are not interested in calling C libraries using some FFI
- That only leaves the existing libraries written in that particular dialect
- They have at best emacs support, which just does not cut it anymore in 2017 for most developers
- Refactoring, Linting and Formatting tools (like in languages like go) are often missing

Community

- The so called “Smug Lisp Weenie”:

  > *Someone who is always eager to let you know that whatever it is you’re doing would be done better in Lisp.*

- Lets just say they have “strong opinions”

Thank You!

Bonus - A bit of History