

WHY CLOJURESCRIPT?

LILY M. GOH

[@LilyMGoh](#)

lily@paren.com

(paren.com)

Why ClojureScript?

(paren)

What is ClojureScript?

Clojure → JVM Bytecode

ClojureScript → JavaScript

(paren)

What is ClojureScript?

Lisp

Functional

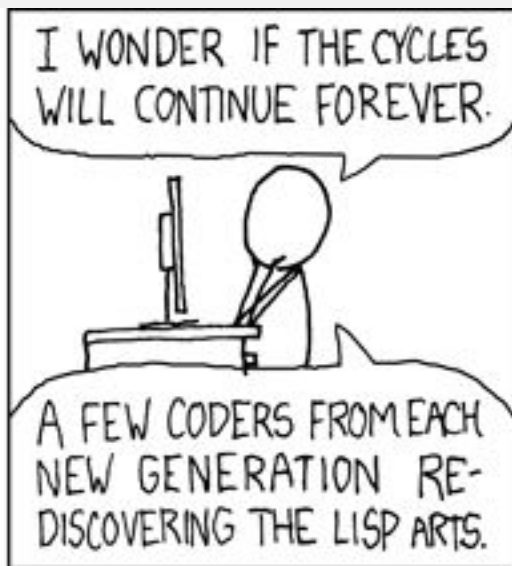
Dynamically-typed

Compiled

(paren)

Lisp

(paren)



Mandatory XKCD

(paren)

LIST Processing

(paren)

Lisp in a Nutshell

JavaScript `func(arg1, arg2)`

ClojureScript `(func arg1 arg2)`

`(paren)`

Lisp in a Nutshell

```
(+ 1 2 3 4)
```

```
(if true  
  "Yes"  
  "No")
```

(paren)

Lisp is Homoiconic

(Code as Data)

(paren)

Lisp is Homoiconic

Code: for, function, if, else, etc

Data: [], {}

(paren)

Lisp is Homoiconic

```
(def my-list (list 1 2 3))  
(map  
  (fn [i] (+ i 1))  
  my-list)
```

(paren)

Lisp is Homoiconic

Code as Data

Easy for Machine

Easy for Human

(paren)

Lisp is Extensible

(paren)

Macro

Function

Input Data

Evaluate Code

Runtime

Macro

Input Raw Code

Manipulate Code

Before Runtime

(paren)

Macro

```
(defmacro hello [name]  
  `(js/alert ~(str "Hello, " name)))
```

```
(macroexpand '(hello "New user"))  
=> (js/alert "Hello, New user")
```

(paren)

Compile v. Run time optimization

(paren)

Functional

(paren)



(paren)

React is functional

(paren)

Immutable.js

```
const { Map } = require('immutable')           (def map1 {:a 1 :b 2 :c 3})
const map1 = Map({ a: 1, b: 2, c: 3 })        (def map2 (assoc map1 :b 50))
const map2 = map1.set('b', 50)                (:b map1) ; 2
map1.get('b') // 2                            (:b map2) ; 50
map2.get('b') // 50

// Convert to JS Object
map1.toObject()
map2.toObject()
```

(paren)

```
const user1 = {  
  firstName: "John",  
  lastName: "Doe"  
}
```

```
const user2 = user;  
user2.firstName = "Jane";
```

```
user1 === user2 // true
```

```
(def user1 {:first-name "John"  
           :last-name "Doe"})
```

```
(def user2  
  (merge  
    user1  
    {:first-name "Jane"}))
```

```
(= user1 user2) ;; false
```

(paren)

```
const styles = {
  container: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center'
  },
  text: {
    fontSize: 32
  }
};
<View style={styles.container}>
  <Text style={styles.text}>
    Hello, World!
  </Text>
</View>
```

```
(def styles
  {:container {:flex 1
               :align-items "center"
               :justify-content "center"}
   :text {:font-size 32}})

[  
  (rn/view {:style (:container styles)})  
  (rn/text {:style (:text styles)})  
  "Hello, World!"]]
```

(paren)

Compiled

(paren)

Google Closure Compiler

(paren)

Google Closure Compiler



(paren)

Google Closure Compiler

```
function hello(name) {  
  alert('Hello, ' + name);  
}
```

```
hello('New user');
```



```
alert("Hello, New user");
```

(paren)

Google Closure Compiler Limitations

Not well-documented

Subset of JS only

Type annotation

Statically access property in object

(paren)

JS is the biggest compile-to-JS language

(paren)

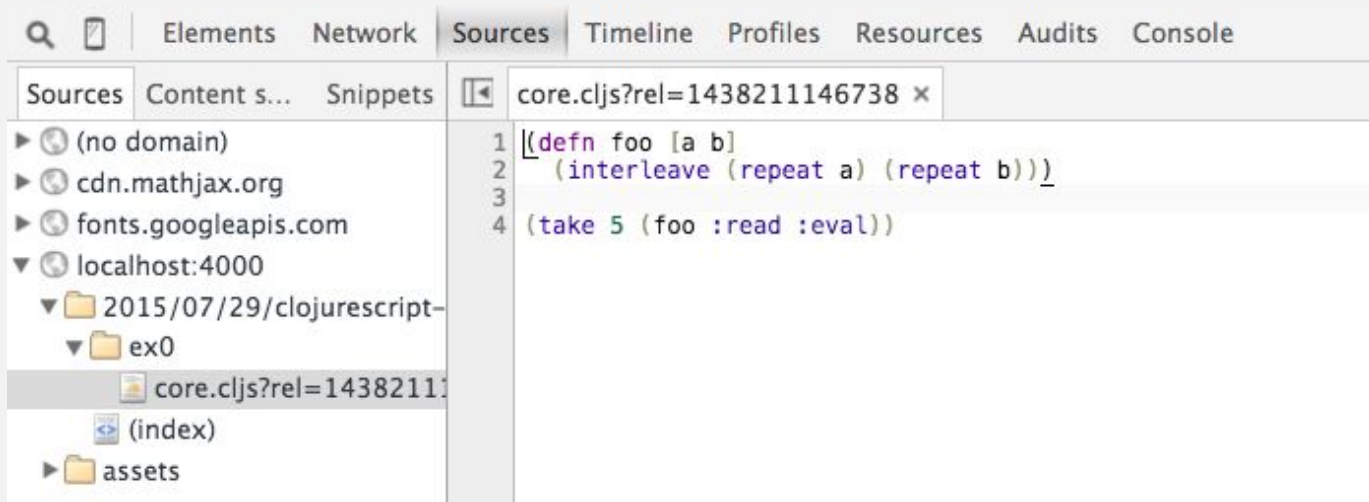
JS Interop

```
alert('Hello World');
```

```
(js/alert "Hello World")
```

(paren)

Source Map



Credit: David Nolen

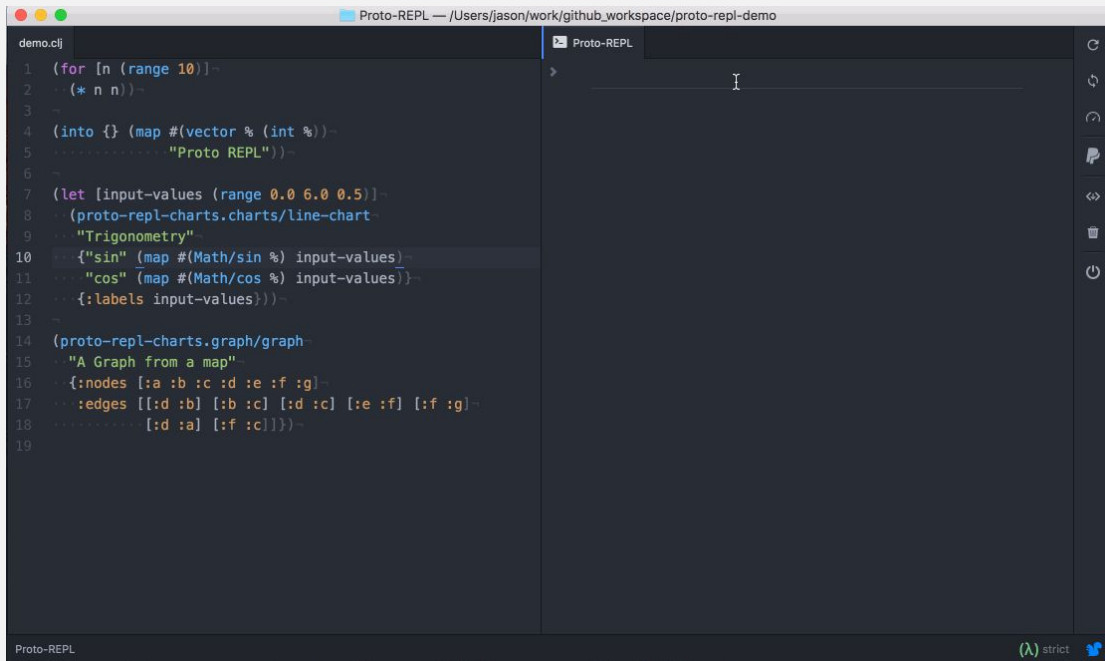
(paren)

REPL Driven Development

Read Eval Print Loop

(paren)

REPL Driven Development



The screenshot shows the Proto-REPL application interface. The left pane is a code editor with a file named 'demo.cj' containing 19 lines of Clojure code. The right pane is the REPL, which is currently empty. The code in the editor is as follows:

```
1 (for [n (range 10)])-  
2   (* n n))-  
3 -  
4 (into {} (map #(vector % (int %))-  
5           "Proto REPL")))-  
6 -  
7 (let [input-values (range 0.0 6.0 0.5)]-  
8     (proto-repl-charts.charts/line-chart-  
9       "Trigonometry"-  
10      {"sin" (map #(Math/sin %) input-values)-  
11        "cos" (map #(Math/cos %) input-values)}}-  
12      {:labels input-values}))-  
13 -  
14 (proto-repl-charts.graph/graph-  
15   "A Graph from a map"-  
16   {:nodes [:a :b :c :d :e :f :g]-  
17    :edges [[:d :b] [:b :c] [:d :c] [:e :f] [:f :g]-  
18            [:d :a] [:f :c]])}-  
19
```


Credit: Proto REPL



How to get started?

(paren)

Structural Editing



The screenshot shows a code editor window titled "paredit.clj - [learnclojure] - project.clj - [--/dev/learnclojure]". The code content is as follows:

```
;; Editing commands

; Slurp will find the next element following your current sexp, and bring it inside.
; Barf will do the opposite - it expels the last element in your current sexp.
; You can do these forwards or backwards.

{(if [condition])
  (do-something)
  (do-something-else)
  (let [my-var (test)]
    (more-stuff)))

; Raise replaces the parent sexp with the current one

{(let [no-longer-needed (my-fn)]
  (if (always-true)
    (do-something)
    (do-something-else)))
```

The editor interface includes a status bar at the bottom with the following information: "141:16", "LF", "UTF-8", "Structural: On", and "341M of 666M".

Credit: cursive-ide.com



How to get started?

Structural Editing

Clojure for the Brave and True

“Simple Made Easy” talk

(paren)

hello@paren.com

LILY M. GOH

lily@paren.com

[@LilyMGoh](#)

(paren)