WILL *Clojure* BRING CLOSURE TO WEB DEVELOPMENT

A PREPRINT

Ali Abdullah A Alshehri,*
Department of Computer Science
University of Arizona
alialshehri@email.arizona.edu

Ray Barnett
Department of Computer Science
University of Arizona
RBarnet6@email.arizona.edu

April 30, 2020

ABSTRACT

Reading Clojure (or any LISP, that is) is sometimes harder than writing it, especially in a large codebase. This is more obvious when compared to Object-oriented programming. After 2 and half years of constant development, Rich Hickey created the Clojure language. It's an open-source, community-driven language. Meaning that anyone is allowed to fork a JIRA ticket from the Git repo and a select few admins will approve of your merges. As a design paradigm, Functional programming adheres to"code is data". It is making programs simpler and cleaner. Clojure performs most of the work at compile-time. This is done by the use of macros. As discussed under section 3.1 control-flow

***K*eywords** First keyword · Second keyword · More

1 Introduction

Clojure is a functional programming language based on Java. There's unfortunately a niche pool of developers who want to use clojure. On the bright side, that makes their skills more in-demand. It's more common to choose any of the more-known languages to build very large projects. Clojure excels in data-science when the data and code have to be kept separate and concurrency is important.

1.1 Hello World

;; A typical entry point of a Clojure program:

```
(defn hello-world []
  (println "Hello World"))
(hello-world)
```

1.2 "something else"

Anything that you feel shows the unique flavor of the language(1-5 lines of code is enough, depending on the language

This code shows how to manipulate code as data is passed around.

```
;; generate list by replacing the elements on the list passed original
(map (fn [form]
       (case form
         1 'one
         + 'plus))
```

*Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

```
    (quote (+ 1 1)))
;; => (plus one one)
```

2 History

Clojure is newer programming language that had its public First stable release in 1.0 May 4, 2009

2.1 Why was the language designed?

often because someone was unhappy with current languages!? Clojure is a good language to start a company/project with. The developers back then did a lot of work to scale Clojure. It took some effort to avoid get that dynamic ripple oriented workflow as well as kind of all that large-scale and distributed computing.

2.2 who designed it?

was it designed committee, by a single individual? Clojure community is a very welcoming group. The embrace of diversity and the ease for the newer unexperinced Clojure community that need on-boarding. *Derek Slager*, Clojure's CTO, said "It's nice to be able to hire from a pool of people that have already embraced that culture so that's really powerful. As you can see we have a nice group of people at our company and I think looking ahead, clojure will keep me up at night." Clojures' development team aim to keep pace in anticipation of growth in the community. According to *D. Slager*, more companies are using it. Clojure's sales-force is dedicated to really get it to be a first-class, industry standard. Potentially, an industry-wide adaptation creates ecosystem effects that are good for everyone (developers, companies, users). E.g. Specific tools. Tools continue development and continue growth so for a business perspective. Tools are something as simple as po

2.3 what is its current status ?

is it alive, dead, or barely moving? It is alive and thriving but at a slow rate. Functional languages are slow to adapt. People who are using Clojure in production code, are solving incredibly complicated problems at incomprehensibly large scale.

3 flow control statments

One needs to understand the different control-flow structures (loops, conditionals, try/catch) are built on top of macros. Clojure big program's are built out of customized macros. E.g. Clojure has emphif-else statements but we can implement build-your-own macro emphunless(condition) do (something) OR we can implement an if-not. Below is the list of most common control-flows that are based on Clojure (note some are inherieted from the JVM exception class)

3.1 1-if then

if (condition): (it's true, then execute this line) (else, execute this line)

```
(defn can-drive
  [myAge]
  (if (>= myAge 18)                    ;; If the statement evaluated
    (println "You are eligible Drive")  ;; true execute the 1st,
    (println "You can't drive")))       ;;or 2nd if false
```

3.1.1 3-if then (do f1 f2)

basically what this is going to do is check for multiple different statements in one condition

conditions

```
(defn can-vote-drive
  [age]
```

```
;; You can perform multiple actions with do
(if (>= age 18)
  (do (println "You are eligible vote")
      (println "You can drive"))
  (println "You can't Vote")))
```

3.1.2 unless (custom opposite of if

;; unless(condition) do (something)

```
(macro def unless [condition & consequence]
    '(if ~ condition ) ;; this is a bitwise negation with an unquote condition
       (do ~@consequence)));;this splices the body of the logical consequence to follow fr
```

input: (unless (even? 3) "3 Is NOT an even number") Output: "3 Is NOT an even number"

3.2 3-WHEN

Is used when you want to do many things if true. It's closest to the switch statment in Java

```
;; WHEN
(defn when-ex
  [switch]
  (when switch
    (println "1st element")
    (println "2nd element")))
```

3.2.1 3-WHEN-NOT

is an alternative of the aforementioned WHEN. It evaluates switch. If it is false, it'll evaluate the "1st element"

3.3 4-COND

It check for multiple conditions

```
(defn passing-grade
  [n]
  (cond
    (> n 60) "Pass"
    else "Fail"
)   )
```

3.3.1 4-CONDP

condp is like cond but it takes a composite predicate expression, and a set of clauses. Here is a recursive function to calculate a list's length.

Input=> (length '(1 2 3 4)) Output=> 4

```
(defn length [lst]
    (condp = lst
        (list) 0 ;if empty list, we reach the base case and result 0
        (+ 1 (length (rest lst)))))  ; default expression
```

3.4 5-loop

Interestingly looping over a list is recursive in Clojure. By using the loop keyword, the loop constructs is a hack such that it's a wrapper around the loop function.

```
(loop [x 1]
  (when (<= x 10) ; base case
    (println x)
    (recur (+ x 2))))
```

loop [1 ... 10] recursive call

```
print 2 loop[ 4 6 8 10]
2 print 4 loop[ 6 8 10 ]
2 4 print 6 loop[ 8 10 ]
2 4 6 print 8 loop[ 10 ]
2 4 6 8 print 10 loop []
;;=> 2 4 6 8 10
```

3.4.1 6-recurs

Note that the last code example had (recur (+ x 2)), which returns feeds the modified argument to its recursive-caller. In other words: The loop is the recursion point for the function recur. The symbols in loop's return values are that of recur's exprs before the next recursive-execution of loop's body. Use recur to feed the new values back into the loop

3.5 7-Exceptions

3.5.1 Throwing Exceptions

The proper way to throw and handle exceptions in Clojure is done with (throw (Exception. errorMessage)) in a conditional (could be if else, cond, when etc...) and then handling the aftermath of errors.

```
(defn stringArray [elements]
  (when (empty? elements)
    (throw (IllegalArgumentException. "elements cannot be empty.")))
  elements)
```

3.5.2 Try / Catch Exceptions

The flow structure and syntax of Exceptions is inspired by Java. Java.lang.ArithmeticException: is found in the Clojure libraries but it's executed and compiled as Java. As shown in the trace stack below:

```
(try (println "1st")
     (println "2nd")
     (/ 1 0)
     (println "LAST LINE")
     (catch Exception e
        ((str "3rd:") .printStackTrace ex)
        (println "ERROR: division by zero. LAST LINE was not executed")))
```

The above code returns the following:

```
;;=> 1st
;;=> 2nd

;;=> 3rd java.lang.ArithmeticException: Divide by zero
    ;; at clojure.lang.Numbers.divide(Numbers.java:163)
    ;; at clojure.lang.Numbers.divide(Numbers.java:3833)

;;=> ERROR: division by zero. LAST LINE was not executed
```

You can also have multiple catch and/or try clauses and it will execute the exception to either errors being thrown and handle it appropriatly.

3.5.3 finally clause

The following code example has multiple catch clauses in addition to a finally clause. Finally always executes whether or not the exception was thrown and handled. It's useful for post-processing or clean-up protocol. E.g. closing a scanner, or a port after closing a program abruptly.

```
(try (f)
    (catch RuntimeException (when (neg? f)
        (println "f must be positive: " f)))
    (catch ArithmeticException e (when (zero? f)
        (println "Trying to divide by zero") )
    (finally (println "I am ALWAYS executed."))))
```

The above code returns the following depending on which input. Let's try 0 and -5:

```
;;=> try(0)
;;=>Trying to divide by zero
;;=>I am ALWAYS executed


;;=> try(-5)
;;=> f must be positive: -5
;;=> I am ALWAYS executed.
```

4 Data types:

Just like any other Computer programming languages, Clojure has the concept of Data Types to express different things using it for type-checking. Clojure has three main type catagories:

4.1 Data types groupings

Types don't need to be explicitly declared. Things like char, string, long, and more complex structers are examples of Data Types They are typically inferred automatically (just like JS, python etc...) and can be optionally specified. The main groupings

- The basic data types
- Abstract data structures
- Miscellaneous types.

4.2 basic data types

Clojure is based internally on the JVM, and therefore any of the standardized types in Java applications are valied for Clojure. In the table below, notice *java.*lang.string vs. *clojure*.lang.Ratio Simple Types Table 4.2.

Name	possible inputs	Description
(type nil)	nil	Null value
java.lang.Boolean	(type true) / (type false)	True/false to navigate conditionals
java.lang.Long	(type 0)	nubmers in the range of 2^32
java.lang.BigInt	(type 1N) / (type 0xffffffffffffffff)	Any big hexadecimal number or a digit followed by N
java.lang.Double	(type 3.14)	Numbers that can include a decimal after 3.xxxxx
clojure.lang.BigDouble	(type 3.14M)	it includes a decimal after 3.xxxxxM followed by M
clojure.lang.Ratio	(type 1/4)	Any (X/Y) any whole digits, and Y != 0
clojure.lang.Keyword	(type :if)	Reserved and cannot be used for variables due to syntax
clojure.lang.symbol	(type 'if)	Won't be evaluated but have to be returned as is
java.lang.String	(type "foo")	A contiguous array of chars. See below
java.lang.Character	(type $\mathring{}$)	Any one char ASCII value

4.3 Data Structures

Data Structures Table 4.3. Clojure comes with native full support for various collections and data structures that can be used Examples include lists, maps, vectors, and sets. However there are differences to Java.

- A vector: is implemented as ordered list of values – arbitrary values can be added into any index of the vector, just like an Array since it's contiguous. Typically, it's preferred since it has less overhead.
- A Map are simple set in essence, they are hash tables with key,values entries.
- A set is an unordered data structure containing values, and can never have duplicates.
- A list is very similar a LinkedList in Java. A list is better if we want to have addToBeginning(), or addToEnd(), ortraverseAllElements() in sequential order.

	Part	
Name	possible inputs	Description
clojure.lang.PersistentVector	(type []) / (type [1,2])	Array or Vector
clojure.lang.PersistentList$EmptyList	(type '())	;Empty LinkedList using pointers
clojure.lang.PersistentList	(type '(1738))	; A linkedList of one node 1738
clojure.lang.PersistentArrayMap	(type)	A hasmap or dictionary of <Key, value>
clojure.lang.PersistentArrayMap	(type)	A hasSet or set. Doesn't allow duplicates

4.4 Other Data Types

Data Structures Table 4.4. Clojure comes with native full support for various data structures that can be used Examples include lists, maps, vectors, and sets. You can even have a collection of collections(e.g. Array[][]). However there are differences to Java.

- A vector: is implemented as ordered list of values – arbitrary values can be added into any index of the vector, just like an Array since it's contiguous. Typically, it's preferred since it has less overhead.
- A Map are simple set in essence, they are hash tables with key,values entries.
- A set is an unordered data structure containing values, and can never have duplicates.
- A list is very similar a LinkedList in Java. A list is better if we want to have addToBeginning(), or addToEnd(), ortraverseAllElements() in sequential order.

	Part	
Name	possible inputs	Description
java.lang.Class	(type Boolean) / (type Integer(2))	; Java's Boolean class.
clojure.core$$_{PLUS}$	(type +) / (type /)	The + OR / operators
clojure.lang.PersistentList	((type (fn [] ()))	; function
user$eval1158$fn$_{1159}$	(type)	A hasmap or dictionary of <Key, value>
java.util.regex.Pattern	(type "[a-z]*")	regular expression

Ideally, in Clojure you want to use a large libraries of functions on primitive data types.This guarantees simplicity in syntax and its emphasis on re-usability.