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

Clojure can be simply defined as a functional programming language that runs on the JVM. It features Lisp-based implementation. 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.

Given its design paradigm, Clojure adheres to"code is data" mentality. 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

Keywords 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.[3] Clojure's most-restrictive feature is the immutability of variables. That makes the language so versatile yet challenging to pick up. The equivalent of a class in Clojure is called a namespace which sets a unique name/scope for any functions. Hence we have to define a class for the JVM machine like:

```
(ns clojure-tut.core
      (:require [clojure.string :as str])
   (:gen-class))
```

1.1 Code Snippets: Hello World

;; A typical entry point of a Clojure program:

```
(def  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.

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

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

```
(def 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

```
(def  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
(def  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

```
(def  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

```
(def  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.

```
(def  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 $2^3 2$
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)̊	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$ $_{P}LUS$	(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.

5 Lexical Structure

The following is the default syntax for building a working function. Clojure uses whitespace as a delimiter to separate arguments[2] Let's define a basic function with arguments.

```
(function arg1 arg2 ... argN) ; Notice that there are no commas.
    (printLn arg1 arg2 ... argN)) ; return value
```

In plain English, it's opening parenthesis, operator, Arg1...N, closing parenthesis. There's no return value, or types declared. The function/operand does NOT care what it is being applied to. You will still face a runtime error if the

[2]Clojure will convert any comma into whitespace as a delimiter.

operation is illogical, but that guarantees simplicity and functions being an argument of another function. In Clojure, functions don't need to have names. Also known as...

5.1 Anonymous Functions

Anonymous functions are used all the time to de-couple and seperate concerns. You can have the keyword *fn*

```
((fn [x] (* x 3)) 8) ;; anonymous function

(def anon-multiplier (fn [x] (* x 3))) ;;a function with inner anonymous class
(anon-multiplier 8)
; => 8

#((* x 3)) 8) ;;standalone anonymous function to be used in a console
```

5.2 Strengths of Clojure

Clojure's lisp-based paradigm gives us functions optimized manipulating the list datastructure. Lists, can be thought of as a graphs. They can also be represented as a sparse matrix in the case of big data!

5.2.1 ideal applications of Clojure

Clojure is usually used for data-analysis and research. Examples:

- It's been used to sequence DNA (like genomics)
- graph theory (such as finding shortest pathways)
- and tabular data (such as that handled by R)
- AI training models. More on that later.

5.3 Weaknesses of Clojure

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. The size and scale of Java projects are usually bigger than 500K lines of code (admittedly a questionable metric).

That makes Clojure a bad option for a new start-up company that expects to maintain the code and it's very likely to NOT find readily, cheap Clojure coders. Functional programming is still a niche used by specific problem domains.

In addition, the company's CEO was rumored to have the following criticism [3]

> "Generically bad advice. right? So what if you choose micro-services instead of monolith architecture? You've got a lot of stuff that you now have to figure out so first and foremost"

Then in a public interview answering a question about scalability for companies

> "What does that even mean? Right micro service it's not like a. canonical dictionary entry for micro-services. The micro service is a project of this exact size that talks about like what, or. how big is a micro service. I don't know. Because there's a lot at stake and of course we, chose closure as the foundational. language for our company".

6 Examples of libraries uses

6.1 Text Generation Using Markovity chain

The paragraph above is relevant to this section. This random, and typo-riddled string was generated by the following code from (markov-chains.core)

[3] This is generated by a FAKE text generator library written in ClojureScript

```
;; generate a random text that's based on a n-ary tree that parsed the n-grams of text. Re
(def Fake-CEO-Text ;construct text model
  (->
    (slurp "./someFile/"Interview script.txt")
    (clojure.string/split #"\s+")
    (collate 10))) ;value of n-grams


;anon function to print the paragraph above.
(->>
  (generate Fake-CEO-Text)
  (take 60) ; how long the generated text will be
  (clojure.string/join " ")) ;joining the text generated recursively.
```

[1]

Mark V. fake-text generator consists of two parts, a table builder (of n-long sentences) and a text generator.

6.2 Java's library

Clojure has direct access to the Java frameworks and libraries. In addition, it provides optional type casting and type hinting. The purpose is to minimize error-throwing calls to Java. [2]

`https://clojure.org/community/contrib_libs`

7 Our short program

This is a visualized physics simulator that aims to quantify and calculate Newtton's gravitational law and returns the path that two objects (of astronomically massive weight) have a force pulling both towards each other.

7.1 output

```
    --------------------------------------------------
Graph A:
       (0m)      (10m)     (20m)    (30m)
    |         |         |        |
    V         V         V        V
1500m:    ........A.........B..........    0s
1425m:    ........A.........B..........    3s
1350m:    .........A........B..........    5s
1275m:    ..........A.......B..........    6s
1200m:    ..........A.......B..........    7s
1125m:    ...........A......B..........    8s
1050m:    ............A.....B..........    9s
975m:    ............A.....B..........    10s
900m:    ............A.....B..........    11s
825m:    .............A....B..........    11s
750m:    ..............A...B..........    12s
675m:    ...............A..B..........    12s
600m:    ...............A..B..........    13s
525m:    ................A.B..........    14s
450m:    .................AB..........    14s
375m:    ..................X..........    15s
300m:    ..................X..........    15s
225m:    ..................X..........    16s
150m:    ..................X..........    16s
75m:    ..................X..........    17s
0m:    ..................X..........    17s
    EarthEarthEarthEarthEarthEarth
```

```
--------------------------------------------------
* Graph A (above) approximately models the positions of two point particles as they descend to Earth.
* The approximate time elapsed is displayed to the right of each line.
* In the event of a collision, the combination of object A and object B is denoted by "X".
* To alter the initial conditions, simply adjust the values of the clearly labeled global variables towa
* Warning! Some initial conditions are invalid and will produce errors.
* This program was composed in Clojure April 4, 2020 by Ray Barnett and Ali Alshehri as a submission for
  CSC372 taught by Professor Collberg.
* Thank you again for all the accomodations.
---------------------------------------------------
```

7.2 Formula physics simulator

The equation can be found coded in GitHub,

The variables that you can work with are

- The weight of 1st body A
- The weight of 2nd body B
- Height. As in how far away from the atmosphere

And here's the mathematical formula:

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

- F: Force.
- G: Gravitational constant.
- m_x: Mass of object x.
- r: Distance between centres of two objects.

References

[1] by A. K. Dewdney. The tinkertoy computer and other machinations. In *The Tinkertoy Computer and Other Machinations Hardcover – 5 Oct. 1993*. IEEE, 1993.

[2] Rich Hickey. The lisp-based java libraries documentation.

[3] Bohdan B. Khomtchouk, Edmund Weitz, Peter D. Karp, and Claes Wahlestedt. How the strengths of lisp-family languages facilitate building complex and flexible bioinformatics applications. volume 19, pages 537–543. Oxford University Press, May 2018. 28040748[pmid].

References

[1] by A. K. Dewdney. The tinkertoy computer and other machinations. In *The Tinkertoy Computer and Other Machinations Hardcover – 5 Oct. 1993*. IEEE, 1993.

[2] Rich Hickey. The lisp-based java libraries documentation.

[3] Bohdan B. Khomtchouk, Edmund Weitz, Peter D. Karp, and Claes Wahlestedt. How the strengths of lisp-family languages facilitate building complex and flexible bioinformatics applications. volume 19, pages 537–543. Oxford University Press, May 2018. 28040748[pmid].