CSE334 – PROGRAMMING LANGUAGES

PROGRAMMING LANGUAGE EVALUATION: CLOJURE

Şerafettin BAL 20150807002

TABLE OF CONTENTS

1. WHAT IS CLOJURE PROGRAMMING?

Clojure programming language is a dynamic, general-purpose, and predominantly functional Lisp programming language. Clojure is a dialect of Lisp but is not a direct descendant of any prior Lisp. Clojure is a programming language from the Lisp family tree. It is used by many companies, large and small, including Amazon, Walmart, and Staples. It is combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming.

Pic.1. *Logo of Clojure*

Clojure is intentionally hosted, in that it compiles to and runs on the runtime of another language, such as the JVM (Java Virtual Machine). Also, Clojure runs in the CLR (Common Language Runtime), and JavaScript platforms and it influenced these programming languages. Also, the name 'Clojure' was created inspired by C#, Lisp, and Java programming languages. (C, L, and J). It is designed for symbiotic and concurrency with the built-in Java platform.

Clojure treats code as data and has a Lisp macro system like other Lisp programming languages. That is, Clojure shares with Lisp the code-as-data philosophy and a powerful macro system.

Clojure advocates invariance and invariant data structures and works to protect this data. It encourages programmers and developers to be open about managing their identities and situations. Focusing on programming with these invariable values and open time progression structures aims to make it easier to develop more robust, especially simultaneous, simple and fast programs. It argues that a reliable and durable programming language should also be useful and fast. The writing system is completely dynamic. But recent efforts have also demanded that gradual writing be applied.

2. HISTORY OF THE CLOJURE PROGRAMMING LANGUAGE

Clojure initially designed in 2005 and released in 2007 by Rich Hickey. Rich Hickey is a computer programmer and prolific speaker. He is the chief technology officer at Cognitect. He has a lot of ideas about programming.

He developed dotLisp, a similar project based on the .NET Framework before Clojure. After Clojure, he also created and designed ClojureScript which is the Extensible Data Notation (EDN) data format, and Datomic distributed database.

Pic.2. *Rich Hickey, author of Clojure*

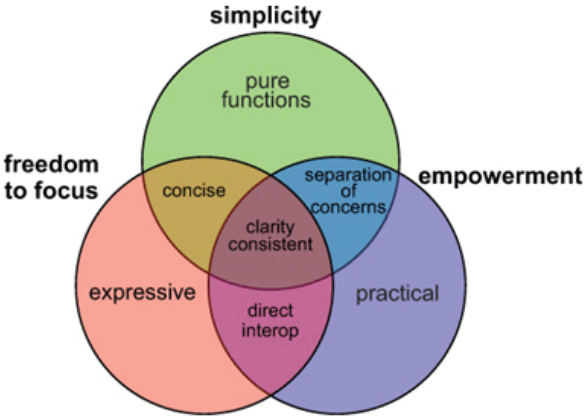
Hickey is an independent software developer over 20 years of experience in many fields of software development. He worked on programming systems, voice and face analysis and fingerprint, database design, exit survey systems, and machine learning.

He spent about 2 years and 5 mounts working on Clojure before releasing Clojure.

3. DESIGN PHILOSOPHY

So, why was Clojure invented? Rick Hickey developed Clojure because he wanted a modern Lisp programming language for functional programming, symbiotic with the established Java, and designed for concurrency.

Clojure's approach is characterized by the concept of identities. This identity concepts are represented as a series of immutable states over time. For this purpose, all workers can operate on states in parallel. That's why reference types are very important for Clojure. These types are interchangeable, each can transfer between states and includes well designed semantics.



Pic.4. *The Clojure Philosophy*

4. WHY CLOJURE?

Simplicity

The Clojure programming language has a very plain, simple structure and the syntax of the language is also very plain. For example, adding the number 2 and 5 in the language is so simple like that: (+2 5) => 7.

For a better understanding, the first version of Lisp written in 1958 used the same semantics as the current ones, like latest version. And there is no syntax evolution on different versions of programming languages, such as Java. The core structure is a very simple and it is a good detail for the programming language.

Java Interoperability

Since it is written on Java Virtual machine (JVM), Clojure gives access to all available Java (or any JVM languages) libraries and their frameworks. So, you can call Java code from Clojure code or vice versa. This is a huge advantage, especially for developing software.

In the following example, the Clojure code runs by using isUpperCase() method which is inside of Character class in Java.

```
(every? #(Character/isUpperCase %) "HELLO")
=> true
```

Read Eval Print Loop (REPL)

REPL is an important thing for Lisp base programmers and it is a tool that enables you to interactively contact with the program you write and evolve it rapidly. And REPL allows you to run any language and any code within the Clojure language.

It is a dynamic programming language and there are very important advantages. For example, you built your web application, but you noticed an error in the info message that will be shown on the screen and change it.

Clojure quickly changes the Java Bytecode and have the change we desire. This saves so much time. And, the other advantage of Clojure is that it can be so flexible and dynamic under a powerful system like JVM. It is dynamic and multi-threaded.

Macro

The macrosystem of Lisp provides you an opportunity to extend the language. Thus, you can get rid of duplicate codes you don't want by writing nice macros.

By using macros, code clutter is avoided. It is more understandable and can be coded simply. And it saves users time. Let's explain Macros with an example.

The macro named if-let * is defined below for the if statement.

```
(defmacro if-let*
  ([bindings then]
   `(if-let* ~bindings ~then nil))
  ([bindings then else]
   (if (seq bindings)
     `(if-let [~(first bindings) ~(second bindings)]
        (if-let* ~(vec (drop 2 bindings)) ~then ~else)
        ~else)
     then)))
```

The code that needs to be nested will be generated by the macro, and the user can simply return the desired outpuy by typing the code below.

```
(if-let* [a 1
          b 2
          c 3]
         (+ a b c))
=> 6
```

Looking at the macro example above, it is possible to give the desired code with less code and to have a better code in large scale project.

Also, using macros, there is no need to wait for synchronization between versions of Clojure. By typing macro, the users can add the missing features of the version used.

Concurrency

Compared to mutable languages, Clojure is a good programming language for concurrency programming. One of the reasons for using Clojure is that its simultaneous programs are designed to be easy, fun and at the same time with much less error rate. Moreover, the language has 4 mutable reference types, and this facilitates our work on state management issues. These are Var, Atom, Agent, and Ref.

Var: Local scope

Atom: Atomic field

Agent: Async programming

Ref: STM (Software Transaction Memory)

ClojureScript

We were taking about ClojureScript when talking about Rich Hickey, albeit a small one. ClojureScript is a compiler for Clojure that targets JavaScript. So, you write Clojure code and finally have an optimized JavaScript code.

It is indispensable for full-stack developers. When working in the fronted and backend, you use a single language, so you have the advantage of motivation and time.

Also, Clojure and ClojureScript can share the same code. Thus, we can write once and use it both on the front and on the server side (backend).

5. CLOJURE IN DIFFERENT PLLATFORMS

Dependencies, Clojure installer and CLI tools

Clojure is compatible with major operating systems such as Mac, Linux, Windows for installation and it is simple to install.

Clojure requires Java. It supports Java LTS releases. But Clojure wants to provide support for all interim versions work as well. This installation can be either an Oracle commercial version or an open source based on OpenJDK.

The Clojure tools require that the java command is on the path or the JAVA_HOME environment variable is set.

Clojure provides command line tools. So, a Clojure REPL can be started, Clojure and Java libraries can be used, and Clojure programs can be started.

After that, Clojure REPL can be started using the clj or clojure command.

Installation on Mac via Homebrew

Install the command line tools with brew the clojure/tools tap:

```
brew install clojure/tools/clojure
```

If Clojure is already installed, then the latest release can upgraded with:

```
brew upgrade clojure/tools/clojure
```

Installation on Linux

Homebrew can used on Linux or Windows with WSL, similar to the Mac installation.

But to install with the Linux script installer, first, ensure that the following dependencies are installed: bash, curl, rlwrap, and Java. Then, use the linux-install script to download and run the install, which will create /usr/local/bin/clj, /usr/local/bin/clojure, and /usr/local/lib/clojure:

```
curl -O https://download.clojure.org/install
chmod +x linux-install-1.10.1.561.sh
sudo ./linux-install-1.10.1.561.sh
```

Installation on Windows

PowerShell 5 or later versions and .NET Core SDK 2.1+ or .NET Framework 4.5+ are installed. Then run:

```
Invoke-Expression (New-Object System.Net.WebClient).DownloadString('https://download.clojure.org/install/win-install-1.10.1.561.ps1')
```

To run Clojure on Windows, if Windows PowerShell are used, invoke via clj or clojure.

If outside powershell like from the old Windows Command Prompt or Git Bash are needed, then try:

```
powershell -command clj
```

With command line argument:

```
powershell -command clj '-J"-Dfile.encoding=UTF-8"'
```

6. EXAMPLE CODES

First is, of course, printing "**Hello, world!**". The **function** *println* invoked with the **argument** *Hello, world!* Invocation of function called applying the function to data in Clojure or other functional programming languages.

```
user=> (println "Hello, world!")
Hello, world!
```

Pic.5. *Hello, World!*

```
user> "h"
"h"

user> 100
100
```

Example code for Macros

Defmacro: To define a macro, *defmacro* used. Like, function a name, docs and argument given it. Note that using quotes ‘ followed by if and not. This is because you do not want to be evaluated when you define the macro.

Without quotes, you will see an exception.

```
user=> (defmacro unless [test then]
          "Evaluates then when test evaluates to be falsey"
          (list 'if (list 'not test)
           then))

user=> (unless false (println "false!!"))
false!!
nil

;; Error
user=> (defmacro unless [test then]
          "Evaluates then when test evaluates to be falsey"
          (list if (list not test)
           then))

CompilerException java.lang.RuntimeException: Unable to resolve symbol: if in this conte
```

Pic.6. *Use of Defmacro in Clojure*

Macroexpand: Macros are replaced with Clojure code before it is evaluated. Too see how it will be replaced without actually evaluating the macro, use *macroexpand.*

```
user=> (macroexpand '(unless false (println "hi")))
(if (not false) (println "hi"))
```

Pic.7. *Use of Macroexpand in Clojure*

Threading Macros: These are macros that helps you to write nested forms in a cleaner and more readable way. “- > " is called thread-first macro. It is first because it is passing down the evaluation of former forms to the first argument of preceding forms.

```
user> (conj (conj (conj [] 1) 2) 3)
[1 2 3]
```

```
user> (-> []
          (conj 1)
          (conj 2)
          (conj 3))
[1 2 3]
```

Pic.8. *Use of Threading Macros in Clojure*

Example code for Delays

Delay: To defer the evaluation of an expression, use delay.

```
user> (def later (do [] (prn "Adding") (+ 1 2)))
"Adding"
#'user/later


user> (def later (delay (do [] (prn "Adding") (+ 1 2))))
#'user/later
```

Pic.9. *Use of Delay in Clojure*

This is the example immediately evaluating an expression. It is return value is bound to **later** var. When use delay, expression is not evaluated immediately, so “Adding” is not printed.

Force: To evaluate and obtain the result of a delayed expression, use force.

```
user> (def later (delay (do [] (prn "Adding") (+ 1 2))))
#'user/later

user> (force later)
"Adding"
3



user> (def later (fn [] (prn "Adding") (+ 1 2) ))

user> (later)
"Adding"
3
user> (def later (delay (do [] (prn "Adding") (+ 1 2))))
#'user/later

user> (force later)
"Adding"
3

user> (force later)
3
```

Pic. 10. *Use of Force in Clojure*

Example code for Futures

Future: If you use future, printing line is evaluated immediately, and after three seconds, println “after sleep” will be evaluated. This is because Clojure puts the expression grouped by future.

```
user=> (do
         (Thread/sleep 3000)
         (println "hello"))

;; Wait for 3 sec and then "hello" is printed
hello
nil
user=> (do
          (future
          (Thread/sleep 3000)
          (println "after sleep"))
          (println "hello"))
hello
nil
after sleep
```

Pic.11. *Use of Future in Clojure*

Realized?: To know if a future is already done, then use realized?. Realized returns true after 5 seconds.

```
user> (def my-future (future (Thread/sleep 5000) ))

user> (repeatedly 6
        (fn []
        (println (realized? my-future))
        (Thread/sleep 1000)))

#'user/my-futurefalse
false
false
false
false
true
```

Pic.12. *Use of Realized? in Clojure*

7. SPECIFIC THINGS OF CLOJURE AND THE CONCLUSION

In Clojure, there is more of an emphasis on concurrent programming, with advanced features, and features types like Atoms, Agents, and Refs. Also Promises, delays, futures, and macros are baked into the language. Although Macro is also available in other Lisp programming languages, it can be said to be more intensive to use in the Clojure language.

Clojure is the most functional of the Lisp languages, even surpassing scheme due to its native lazy evaluation. And Clojure feature immutable data types.

Finally, Clojure has a strong emphasis on polymorphism without objects as we usually think of them. Multi-methods, multi-threads and protocols allow this to be done in Lisp programming way.