Contents

2. Static vs dynamic typing. duck typed?

Clojure, like Lisp, is dynamically typed[1]. Lisp was a pioneer in adopting dynamic typing (Paul Graham's website). There are no mutable states in Clojure (Fogus & Houser, 2014), so the following example is a work-around to illustrate dynamic typing.

```
usr=> (defn foo [x y] (/ x y)) ;; x / y
usr=> (foo \a 1) ;; char 'a' and int 1
ClassCastException java.lang.Character cannot be
cast to java.lang.Number  clojure.lang.Numbers.
```

```
divide (Numbers.java:159)
;; So the exception came from divide operator,
;; not the type checking of foo()
;; foo() does not require args to be certain types
;; so clojure is ducked typed as well.
```

As shown in the code, Clojure is **duck typed** as well.

[1] https://www.quora.com/Is-Scala-faster-than-Clojure. Michael Klishin seems to be an experienced developer of Clojure.

3. Untyped vs weakly typed vs strongly typed

Clojure is strongly typed, at least stronger than Java; see the tests below.

```
;; Clojure is typed, see the cast exception
usr=> (/ \a 1)
ClassCastException java.lang.Character cannot be cast to
java.lang.Number  clojure.lang.Numbers.divide
(Numbers.java:159)
;; Clojure being unable to cast character into int
;; does not do nonsense math
;; => Stronger than Java

usr=> (/ 1 4.2)
0.23809523809523808
;; Coerced 1, an int, into floating point
;; => weaker than Ada
```

6. Reflection (computational and structural)

Clojure has an API for clojure.reflect (Halloway, 2015). It mainly reflects on the host types, such as the class of an instance. Clojure's reflection is computational. There are two ways to do reflection:

- Reflects on a class that implements TypeReference (type-reflect).
- Get the class of an instance, or reflect on the class itself.

Reflection can provide a map with keys like `:bases` and `:members`.

```
;; reflect on the number 42
;; getting the base classes of 42
```

```
clojure.reflect=> ((reflect 42) :bases)
;; returns a set of java classes
#{java.lang.Comparable java.lang.Number}
```

See the API.

7. Hazards

Which ones can you make? C has aliasing, dangling references, and memory leaks. Java has only aliasing.

Answer

Aliasing

- Normally, Clojure var's are immutable and do not have states; aliasing does not happen.
- If interop'ing with Java, Clojure can invoke method calls on Java objects, and aliasing is indeed possible.

```
(def ^:dynamic x (java.awt.Point. 1 1))
;; y points to the same object as x
(def ^:dynamic y x)

(defn aliaspt []
  ;; changing y
  (. y setLocation 2 2)
  ;; altered the state of x
  ;; returns x.toString()
  (. x toString))
```

Output:

```
user> (aliaspt)
"java.awt.Point[x=2,y=2]"
```

Other hazards

I searched around and found none. Since Clojure is hosted on Java, it is safe to assume that it does not suffer from dangling references or memory leaks.

---

8. Static and/or dynamic scope.

Clojure supports both static and dynamic scopes (Fogus & Houser, 2014).

`let`

`let` defines locals of static scope.

```
(def x 3)           ;; binds global var x to 3
;; (let [bindings*] exprs*)
;; symbols are resolved within
(let [x 1]          ;; binds x with 1
  (let [x 42]       ;; binds x with 42
    (println x))    ;; in the inner let block
  (println x))      ;; in the outer let block
;; global var unchanged
(println x)
```

Output:

```
42
1
3
```

From (Fogus & Houser, 2014)(Ch.10.6.4).

`binding`

`binding` takes a dynamic var, and the new binding affects the call stack .

Application of dynamic scoping in Clojure

`binding` can allow a var to be shared safely between threads; each thread pushes / pops its own local binding of the same var, leaving the global var unchanged, nor does its local binding affect other threads. The following example only shows one simple use case of binding.

But

It is not within the scope of this report to present the advantage of dynamic scoping in a concurrent context. The code examples are either not illustrative or broken in clojuredoc due to API change among different versions of Clojure.

```
(def y 3)
#'clojure-noob.core/y
usr => (binding [y 3] (println y))
IllegalStateException Can't dynamically bind
non-dynamic var: clojure-noob.core/y
clojure.lang.Var.pushThreadBindings (Var.java:320)

;; has to be declared dynamic
usr => (def ^:dynamic x 1)
usr => (binding [x 2] (println x))
2   ;; prints x is 2
nil ;; binding returns
usr => (println x)
1   ;; outside binding, x is still 1
```

From clojuredoc and (Fogus & Houser, 2014)(Ch.10.6).

---

9. parameter passing mechanisms

Answer

By value

Like Java, Clojure passes arguments by value; in the case of references, they are passed by references pointing to objects (Topolnik, 2015). One way to test it is to use `identical?`, which decides if the two arguments are the same object (dpritchett, 2015).

```
;; define a global z with a vector
usr => (def z [1])
#'clojure-noob.core/z
;; foo takes a parameter
;; foo uses identical? to check
usr => (defn foo [a] (identical? a z))
#'clojure-noob.core/foo
usr => (foo z)
;; the z passed in and the global z are the same obj
true
```

10. evaluation strategy

Clojure is partly a lazy language (Fogus & Houser, 2014).

Strict

Normally, Clojure uses *eager* evaluation, which starts from the innermost parenthesis and works its way outwards. Notice that in Clojure, the precedence of operators is **not** a problem because of all the ((((((...))))))'s.

```
;; precedence is not a problem here
usr => (* 7 ( + 4 2))
42
```

Example with side-effect to show strict evaluation

Try:

```
;; a is immutable tho
(def a 3)

(defn sqr [x]
  (println "sqr")
  (* x x))

(defn dbl [x]
  (println "dbl")
  (+ x x))

(defn main []
  (println (sqr (dbl a))))
```

Output:

```
;; would have printed sqr first
;; if eager evaluation is used
user> (main)
dbl
sqr
36
nil
```

Adapted from homework.

Lazy

In the above example, if clojure used call-by-name, then Clojure would have printed two `dbl`'s because function `dbl` should be called twice on the line `(* x x)`. (Clojure data is basically immutable, so there is no eazy way to increment that global `a`.)

When dealing with sequence types, Clojure uses lazy evaluation. For instance, Clojure only "takes" what it needs:

```
;; (range) is a sequence of 0 to infinite int's
;; it did not compute the entire list
usr=> (take 2 (range))
;; universe is still intact because Clojure is lazy
(0 1)
```

---

11. Built in list comprehensions

Yes Clojure has list comprehensions using macro `for`.

```
;; (for sequence-exprs body-expr)
;; python equiv of for x in range(3):
usr => (for [x (range 3)] (* x x))
;; returns a new list
(0 1 4)
```

Do some combinatorics:

```
user> (for [x [1 2 3] y [4 5 6]] [x y])
([1 4] [1 5] [1 6] [2 4] [2 5] [2 6] [3 4] [3 5] [3 6])
```

Nested:

```
user> (for [x (for [y (range 3)] (+ y 100))] (* x x))
(10000 10201 10404)
```

See clojuredoc.

12. Higher order functions. Haskell supports these in many ways. C lets you pass functions into a function which is higher order but that's it.

A higher order function is defined such that it takes functions as arguments, and / or it returns a function as a result (Fogus & Houser, 2014) (Ch.7.1.2).

Functions as arguments

There are three important high order functions in Clojure (Fogus & Houser, 2014) (Ch 6.3.4).

- `map`: applies the function passed in as argument to the provided sequence, and returns another sequence as the result.
- `reduce`: applies the given function to the first two items in the collection; it then take the intermediate result and apply the function with the third item and so forth.
- `filter`: applies the function to every item in the collection; if function returns false for an item, this item is filtered out.

```
;; =============== map ===================
;; dec decreases the value of argument by 1;
;; dec as a function, is one of the two arguments
;; map applies dec to every element in [0 42]
;; A new sequence is returned.
user=> (map dec [0 42])
(-1 41)

;; using map to get columns of a 2D array
user=> (map vector [:x :1] [:y :2] [:z])
;; map stops when one of the sequence runs out
([:x :y :z]) ;; so no [:1 :2] here

;; =============== reduce ================
;; Go Gauss!
;; range returns a lazy sequence from 1 to 100, inclusive
;; Step 1: 1 + 2 = 3
;; Step 2: 3 + 3 = 6
;; Step 3: 6 + 4 = 10 and so forth
;; The sequence is "reduced" to 5050, the sum of all items
user=> (reduce + (range 1 101))
5050
```

```
;; ============= filter ==================
;; even? is a function
user=> (filter even? (range 41 43))
;; between 41 and 42, inclusive
(42) ;; 42 is even!
```

Adapted from (dpritchett, 2015).

Functions as return values

Clojure has three important functions in this category (Fogus & Houser, 2014):

- `comp` takes a set of functions and return a composition function. The returned function will apply the set of functions from the right most to the left.
- `partial` takes a function f that expects n arguments, and x arguments of f. `partial f args`. Returns a fn that takes y arguments, such that $x + y = n$.
- `complement` returns the opposite truth value for the argument function passed in.

It is also possible to define a new function that returns a function.

```
;; ============== comp ====================
;; First, -84/2 = -42
;; Second, -(-42)
;; Applied from right to left
user=> ((comp - /) -84 2)
42

;; ============== partial ================
;; Partial returns a fn: "4" + ?
;; then fn found its other arg to be "2"
;; In Haskell you get the currying for free tho.
user=> ((partial str "4") "2")
"42"

;; ============= complement ==================
;; (complement odd?) effectively returns an even? function
user=> (filter (complement odd?) (range 41 43))
(42)
```

Definitions adapted from clojuredoc.

---

13. infinite lists. Haskell supports these.

Yes Clojure has those, too. With lazy evaluation, one is encouraged to `take` a finite number of items from the infinite list without breaking the universe.

We talk about three flavors of infinity:

1. `(range)`, the equivalent of Haskell's `[1..]`.
2. stream
3. list comprehension

(range)

```
;; (range) returns a lazy sequence,
;; staring from the default 0 to infinity
;; Run this and see what infinity means
usr => (println (range))
;; or, just take the first two in the coll
usr => (take 2 (range))
(0 1)
;; starting infinity from i
usr => #(drop % (range))
```

See clojuredoc

Stream

Stream is a function that returns an infinite list.

```
;; the recursive calls go on forever
(defn fib [m n]
  ;; need to tell clojure to be lazy about this
  (cons m (lazy-seq (fib n (+ m n)))))

(defn fibs []
  (fib 1 1))
```

Output:

```
user> (take 8 (fibs))
(1 1 2 3 5 8 13 21)
```

Inspired by CSCI 208 class notes; see stackoverflow for usage of `lazy-seq`.

List comprehension

```
;; (range) still does the trick
user> (take 4 (for [x (range)] (* x x)))
(0 1 4 9)
```

14. Anonymous functions

As a functional language, Clojure uses a special form to build anonymous, or unnamed, functions (Fogus & Houser, 2014).

```
;; This function has no name
((fn [n]      ;; a vector of function params, n
  (* 3 n))             ;; return value = 3 * n
 2)                   ;; passing in param
```

Output:

```
6                        ;; return value
```

There is also a reader feature to define in-place functions using `#()`.

```
usr> (#(* 3 %) 2)
;; The % implicitly declares the accepted argument
```

Adapted from (trptcolin, 2015) and (Fogus & Houser, 2014).

15. polymorphism. Most languages have some support for this. Haskell has a ton of it.

Answer: Clojure has a ton of it, too.

We covered three types of polymorphism:

1. adhoc / overloading polymorphism
2. Inclusion polymorphism
3. parametric polymorphism

Ad hoc / Overloading

Many functions, same name. In Clojure, a dispatch function, `greeting` in this case, is called first, then the right method is called based on the dispatch value.

```
;; greeting is a dispatch function
(defmulti greeting
  "The multimethod greeting"
  ;; dispatch depends on the key "language"
  ;; the param is a map, which can be considered a function and the
  ;; argument given is "language"
  (fn [x] (x "language")))
;; lots of greeting here, so ad hoc polymorphism
(defmethod greeting "English"  [_]
  "Returns the appropriete greeting"
  "Hello!")
(defmethod greeting "Deutsch"  [_]
  "Hallo!")
(defmethod greeting :default params [_]
  (throw (IllegalArgumentException.
  (str "Kein " (params "language")))))


;; defining maps here
;; the combination "language": "English"
;; makes the dispatcher calls the appropriate method
(def english-map { "id" "1" "language" "English"})
(def deutsch-map {"id" "2" "language" "Deutsch"})
(def espano-map {"id" "3" "language" "Espano"})
```

Output:

```
clojure-noob.core> (greeting deutsch-map)
"Hallo!"
clojure-noob.core> (greeting espano-map)
IllegalArgumentException Kein Espano
clojure-noob.core/eval3459/fn--3460 (core.clj:27)
```

Inclusion / Subclass / Subtype / Inheritance

Clojure allows inheritance relationships among data types or even among symbols (Fogus & Houser, 2014). `defrecord` defines a Cartesian product (class), and `defprotocol` defines Java's equivalence of a interface, without implementation.

```clojure
;; defprotocol does not define class members like name
(defprotocol Walkable
  (walk [this]))
;; Walkable -> Human
(defrecord Human [name]
  ;; Java equiv of implements Walkable
  Walkable
  ;; defining the concrete method
  (walk [this] "Human walks"))
;; Walkable -> Dog
(defrecord Dog [name]
  Walkable
  (walk [this] "Woof"))
```

Output:

```clojure
user> (walk (Dog. "name"))
"Woof"
user> (walk (Human. "the killers"))
"Human walks"
```

Parametric Polymorphism

Functions that can be applied on different types. Clojure has it.

```clojure
(defn head
 "clojure collection methods are
  excellent examples of parametric functions"
  [coll]
   ;; returns head of the coll.
   (first coll))

;; map, returns the first key-value pair
user> (head {:a "a", :b "b"})
[:a "a"]
;; vector
user> (head [1,2,3])
1
;; list
user> (head `(1 2 3))
1
```

Also see: 8thlight, clojuredoc/defmulti, clojuredoc/defprotocol.

16. overloading of method names or operators. Java allows overloaded method names.

In Clojure, operators are plain functions (Fogus & Houser, 2014) (p. 13). Overloading is possible for both methods and operators. In practise, it is better to use multimethods than simply overloading functions, especially basic operators like `+` (see clojure.org).

```
;; attempt to overload + operator
user> (defn + [x] 0)
;; compiler gives a warning
WARNING: + already refers to: #'clojure.core/+ in
namespace: user, being replaced by: #'user/+
#'user/+
;; but the evil definition is used anyways
user> (+ 3)
0
;; what an awful thing to do
```

Source: my humble concoction.

Overloading methods

Arity overloading

Clojure also comes with overloaded methods. Clojure does arity overloading within a single function definition. Below is a snippet of the source code for + operator in `clojure.core`.

```
(defn +
  "Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'"
  {:inline (nary-inline 'add 'unchecked_add)
   :inline-arities >1?
   :added "1.2"}
  ([] 0)            ; no argument
  ([x] (cast Number x)) ; one argument
  ([x y] (. clojure.lang.Numbers (add x y))) ; 2
  ([x y & more]                              ; or more
     (reduce1 + (+ x y) more)))
```

See github

Ad hoc polymorphism

See question 15, discussion on `defrecord` and `defprotocol`.

17. regular expression support in a library? Java has a regular expression library (so does Python).

In Clojure, a regular expressions is a string prefixed with a hashtag (Rathore, 2011). Clojure's regular expression uses Java's regex. Regular expression is listed as one of the scalar literals in (Rathore, 2011), and called "first-class data type" in (Fogus & Houser, 2014).

```
clojure-noob.core> (class #"gneiss")
java.util.regex.Pattern
```

There are mainly four things we do with regular expressions:

1. recognize
2. search
3. capture / backreference
4. replace

Recognize

Clojure has `re-matches` that returns true only if the entire given string matches the regular expression given.

```
user> (re-matches #"[+-]?[0]*[0-9]+" "-3")
"-3"
user> (re-matches #"[+-]?[0]*[0-9]+" "-3.14")
nil
user> (re-matches #"[0-4]?[3-9]+(6*[2-4a-e]+)*" "176a3b62")
["176a3b62" "62"]
```

Search

Search inside a string for a regular expression. In general, languages can adopt one of the two options for searching:

1. greedy - returns the longest portion of the string that matches.

2. non-greedy - returns the first part of the string that matches. (clojure's choice)

Clojure has `re-find` that does searching.

```
;; always the first match
user> (re-find #"fo*" "fooofooooooooo")
"fooo"
;; so it does non-greedy
user> (re-find #"fo*" "fooofo")
"fooo"
```

... or use `re-seq` to get a lazy sequence of all finds

```
user> (re-seq #"[0-4]?[3-9]+(6*[2-4a-e]+)*" "1336b09e176a3b62047b62")
(["1336b" "b"] ["09e" "e"] ["176a3b62" "62"] ["047b62" "62"])
```

Replace

Clojure has `replace` in `clojure.string` that finds all instances of match and replaces them with a new string.

```
user> (clojure.string/replace "Hello World" #"o" #(str % %))
"Helloo Woorld"
```

See clojuredoc.

18. Give a brief history of the language

... and its current state (counts as one question). This question does not require additional code examples or arguments to back up your reasoning. (Just a reference)

- When was it created (range of years or a year)
- Who created it (person, committee, group at MIT, ...)?
- Is it compiled or interpreted? (Or can you do both with it?) If it's somewhere in between, say so and explain.
- Was it created for a purpose? What purpose?
- Is anybody hiring programmers for this language right now? Who? What kind of jobs?

Answers

- Clojure was created by Rich Hickey (richhickey) in 2007. It took Hickey about two years to write it before the initial release (see his github repo).
- Clojure compiles down to Java bytecode see a video of Hickey's.
- Built from Hickey's personal revelations in working as a software engineer, Clojure was created to be the following because C++ / C# / Java are not good enough (see clojure's rationale).
  - A functional language, especially Lisp-like.
  - symbiotic with a popular platform for portability and practicality.
  - concurrency-friendly.
- Yes. People are hiring for Clojure, and the pay is pretty good: numbers. Clojure jobs come from start-ups like Cognitect, and of course there are more. A search on LinkedIn revealed a variety of job opportunities (see linkedin) in areas of web development, data science and more.

19. Implicitly or explicitly typed?

For example, In C all declarations are explicitly typed (globals and parameters and local variables are declared with types, struct fields are declared with types, even functions have explicit return types). In Python or Ocaml, the answer is very different.

Answer

Clojure is implicitly typed.

Declaraing a `var`:

```
;; not type info involved
user> (def a 42)
```

A function:

```
;; no type info for param or return type
user> (defn foo [x] (+ 2 x))
```

Type hint

Clojure does allow type hints. Type hints can be useful for improving performance when interopping with Java (Emerick, Carper, & Grand, 2012).

```
(defn length
  ;; java's String class
  [^String s]
  (.length s ))
```

Output:

```
user> (length "dfd")
3
user> (length [1])
;; look, type error
ClassCastException clojure.lang.PersistentVector cannot be cast
to java.lang.String  user/length (test.clj:77)
```

23. For languages with both a stack and a heap

. . . what goes where? (In Java, arrays are all on the heap. primitive contents are actually inside the array. object contents are somewhere else on the heap. In C, array are where ever you want them. Contents are in the array or out on the heap as you like).

Answer:

Since Clojure is written in Java (and in Clojure of course) and runs on JVM, it's reasonable that Clojure uses the same memory allocation strategy as Java, as stated above. As an indirect evidence, *Clojure Programming* says that *many thousands* of agent objects can be created on the heap (Emerick et al., 2012). I did not find any keyword or special form to specify where things should be stored.

24. Is string a primitive type?

(In C++ it is, in Java, C, Haskell, . . . it is not). What math operations can you do on strings? (s+s, s*3) What other operations are provided for strings? (none, substring, index of). Is there a concept of a terminating character like the NULL in C?

Answers

- According to its inventor, Clojure String is one of the *atomic data types*; an array of characters is not the same as a string in Clojure (see this video of Hickey's).

- Clojure does not do string math (+, *), otherwise `ClassCastException` will occur.
- There is a rich API for string manipulation, including:
  - reverse
  - replace
  - join
  - index-of
  - subs (substring)
- No there is no such thing as not having null-terminated the String and the program seg faults.

Exmaple of substring

```
user> (subs "01234" 1 3)
"12"
```

See clojure source code.

25. What math operations can you do on numbers? Is this different for characters and other integer-like types? Does it allow math that makes no sense?

Clojure's core (link) supports: `+ - * / inc dec quot rem min max`. It also supports comparisons: `== < <= > >= zero? pos? neg?`. Outside the core, clojure.math.numeric-tower (link) provides exponents, ceiling, floor functions and more. There is also the clojure.math.combinatorics (link) to do discrete maths.

Example of nonsense math

```
;; \a is the character 'a'
user> (+ \a 1)
ClassCastException
```

No Clojure does not do nonsense math.

26. Can you directly access or manipulate the bits of an integer value? What operations are possible?

Yes.

According to clojure.org (link), it can do

- bit-and
- bit-or
- bit-xor
- bit-not
- bit-shift-right
- bit-shift-left.

The following example uses bitwise and & bitwise shift right.

```
;; from a CSCI 206 lab
;; get_byte - Extract byte n from word x
;; example: get_byte(0x12345678, 1) = 0x56
(defn get-byte
  [x n]
  (format "0x%x"
        (bit-and
         (bit-shift-right x (* 8 n))
         0xff)))
user> (get-byte 0xdeadbeef 0)
"0xef"  ;; endianness
```

See clojuredoc, bit-shift-right (link).

30. What about dangling else? Is it a problem? If yes, how does it handle it? if not, why is it not a problem? It is a problem in both C and Java but not in Haskell

Answer: no

Since Clojure is fully-paranthesized, there is no such problem as dangling else; every `else` knows which `if` it belongs to.

```
(defn funk []
  (if true
    (if false
      ;; execute this line if true
      (println "nested")
      ;; execute this line if false
      ;; there is no keyword else anyways....
      (println "this else knows...."))
    ;; meanwhile outside the parenthesis:
    (println "Am I getting printed?))
```

Output:

```
user> (funk)
this else knows....
nil
```

More about `if`

Since Clojure has all expressions, `if` always returns something. The argument in case of `else` condition is optional. Specifically, if nothing is supplied to the `else` condition, Clojure defaults to return nil. It is a problem in that user does not have to specify anything for the dangling else; on the other hand, something is always returned.

```
;; prototype: (if test then else?)
;; then clause = "try dangling"
;; else? is missing here
user> (if false "try dangling")
nil ;; default
```

See clojure.core.

31. What is the order or precedence for all the math operators?

. . . relational operators? any other operators? This requires an answer but not code or an argument.

Answer

This question is **not a problem** in Clojure; it does not have precedence rules because it is fully parenthesized (Emerick et al., 2012).

39. Does it use short circuit evaluation? Are these operators with options or not as you like?

Clojure does short-circuit evaluation for `and` and `or`. Google searches yielded no trace of short-circuiting being optional.

```
user> (or true
    (do (println "no short-circuit")))
true
;; nothing prints so does short circuit
;; the operator and does the same thing
;; (and false ....) short circuits
```

Seestackoverflow (link).

40. What is a BNF grammar for it?

Clojure, like Lisp, is known for having very simple grammar. clojure official site (link) describes how reader deals with Clojure grammar through various forms.

The closest thing to BNF for Clojure is attached (deleted some for succinctness), from github/ccw (link).

```
/*
 * Lexer part
 */

SPECIAL_FORM: 'def' | 'if' | 'do' | 'let' | 'quote' | 'var' | 'fn' | 'loop' |
            'recur' | 'throw' | 'try' | 'monitor-enter' | 'monitor-exit' |
            'new' | 'set!' | '.'
    ;

STRING
    :  '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;
```

```
NUMBER: '-'? '0'..'9'+ ('.' '0'..'9'+)? (('e'|'E') '-'? '0'..'9'+)?
    ;

CHARACTER:
        '\\newline'
    |   '\\space'
    |   '\\tab'
    |   '\\u' HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT
    |   BACKSLASH .   // TODO : is it correct to allow anything ?
    ;

literal:
        STRING
    |   NUMBER
    |   CHARACTER
    |   NIL
    |   BOOLEAN
    |   KEYWORD
    ;

KEYWORD:
        ':' (':')? SYMBOL_REST+ ('/' SYMBOL_REST+)?
    ;

/*
 * Parser part
 */

form    :
     {this.inLambda}? LAMBDA_ARG
    |    literal
    |   COMMENT
    |   AMPERSAND
    |   macroForm
    |   dispatchMacroForm
    |   set
    ;

vector:  LEFT_SQUARE_BRACKET form* RIGHT_SQUARE_BRACKET
    ;

map:    LEFT_CURLY_BRACKET (form form)* RIGHT_CURLY_BRACKET
    ;
```

dpritchett. (2015). API for clojure.core - clojure v1.7 (stable). *Accessed Online*

from Https:// Clojuredocs.org/ Clojure.core.

Emerick, C., Carper, B., & Grand, C. (2012). *Clojure programming.* " O'Reilly Media, Inc."

Fogus, M., & Houser, C. (2014). *The joy of clojure.* Manning Publ.

Halloway, S. (2015). Clojure reflect. *Accessed Online from Https:// Clojure.github.io/ Clojure/ Clojure.reflect-Api.html.*

Rathore, A. (2011). *Clojure in action.* Manning Publications Co.

Topolnik, M. (2015). Re: Clojure question -google groups. *Accessed Online from Https:// Groups.google.com/ Forum/ #!Topic/ Clojure/ 5LosIrRXnOY.*

trptcolin. (2015). Clojure koans. *Accessed Online from Https:// Github.com/ Functional-Koans/ Clojure-Koans.*