# CSCI 208 Project

Clojure answered

Xiaoying Pu

## Contents

# Front Matter

This project is also on github (link), with *incomprehensive* code examples in `test.clj`.

## How to run code

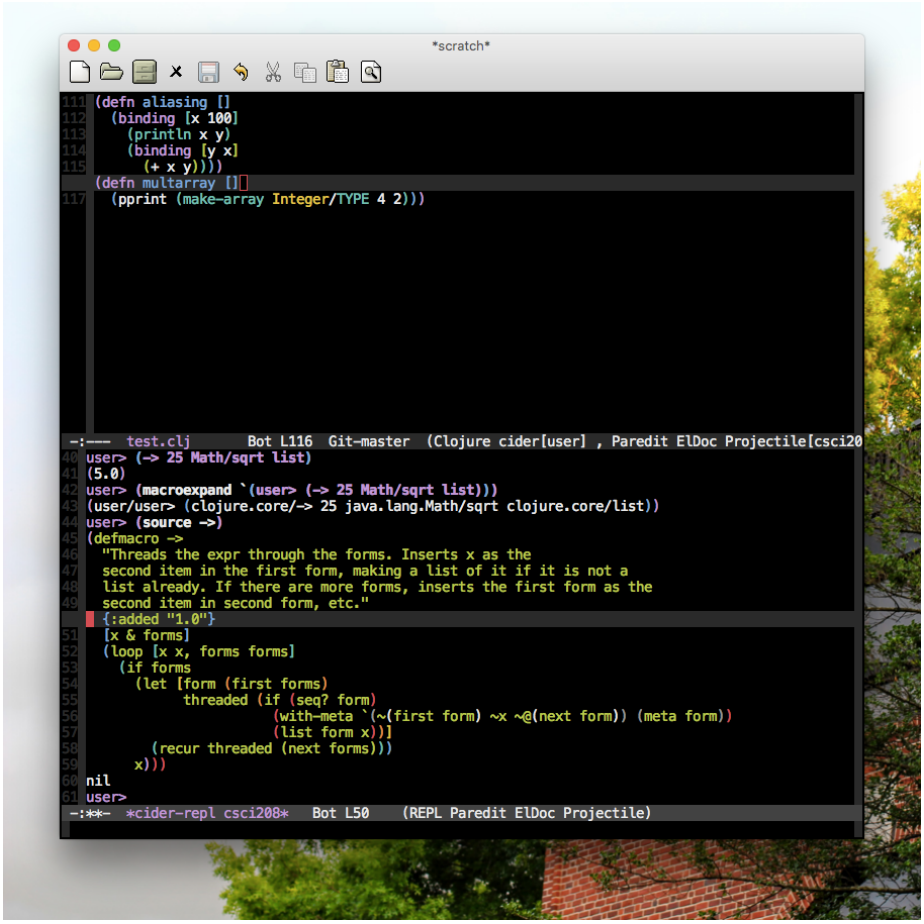


Figure 1: Emacs with REPL running on the lower pane. You can just enter code or call compiled functions at the prompt.

braveclojure (link) teaches how to set up your emacs to work with Clojure conveniently. You can compile Clojure code and run REPL within emacs. In summary:

1. Install Java Runtime Environment (JRE).

2. Install Clojure (link).
3. Install Leiningen (link) and get the dependencies right.
4. Install cider (link), a plug-in for emacs.
5. Open source (if any) file in emacs.
6. `M-x` and choose `cider-jack-in` to start REPL.
7. `C-c C-k` in the code buffer to compile current file.
8. Paste an expression at the prompt in REPL, or paste a function into a file, compile and call the function from prompt.

## This report

- there are a lot of inline hyperlinks as citations; those links unfortunately are not underlined, nor are they blue.
- if you have the Haskell and LaTeX dependencies figured out, you can go to the `project` directory and compile the PDF from source!

```
pandoc -s *.md --bibliography biblio.bib --csl apa.csl
--toc  -o report.pdf
```


Figure 2: For the brave and true

## Phase 1: Hello World

```
(ns p1.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))

; This is Clojure
; yes this is generated by lein, so no ref
```

Compile and run:

make sure that you have started the project with `lein`, and then do `$ lein run` in the terminal.

Output:

```
Hello, World!
```

`lein`, Leiningen, is here.

## Phase 2: Paradigms and Features

What paradigms is your language? What features of the language support your answer? Match specific features to specific paradigms.

### Answers

Clojure is a dynamic languange, and it belongs to the following paradigms: functional, (beyond) object-oriented and concurrent programming.

### Dynamic

Clojure, like its parent Lisp, is first and foremost a dynamic language (Hickey, 2015a). Our text may not have listed being dynamic as one of the paradigms, but it is one of the most important features of Clojure. A dynamic language determines many more things at run-time than in compile time (Rathore, 2011).

- Clojure does not require type declaration [rathore2011clojure].
- Clojure has an `eval` function to load code for execution in run-time, a feature most static languages lack (Hickey, 2015a).
- REPL, Read-Eval-Print-Loop, is the primary interface for programming in Clojure. It is a console for entering and executing commands (Hickey, 2015a).
- Clojure is compiled, but REPL automatically "compiles to JVM bycode on the fly" (Hickey, 2015a).

### Functional programming

According to the textbook, a functional language is based on functions over types such as lists and trees. Clojure is a dialect of Lisp, a classical functional language. Clojure is suited for developing significant programs without having mutable variables, as it possesses, but not limited to, the following characteristics:

1. Clojure functions are first-class functions. Like the more familiar integer or boolean objects, Clojure functions can be created, stored, used as function arguments and returned by other functions (Fogus & Houser, 2014). In addition, it is easy to use high order functions like `map, reduce, filter` (Rathore, 2011).
2. Clojure has a fine selection of *immutable*, persistent data structures. Imperative languages feature the use of variables and manipulation of memory contents. By comparison, the values in Clojure is immutable (Fogus & Houser, 2014); a var in Clojure is the closest thing to a variable in imperative languages (Fogus & Houser, 2014).
3. *Lazy eval*: Clojure sequences are lazy, meaning that the elements are only computed when needed. It is thus possible to create list of infinite length (Rathore, 2011).
4. Anonymous functions: the keyword `fn` mentioned above created a function without a name. It takes a special form in Clojure syntax, because it is not defined in terms of functions, types or macros (Fogus & Houser, 2014).
5. Recursive looping: since Clojure does not employ mutable local variable, its implementation of `for` or `while` loop is different from imperative languages with changing states (Hickey, 2015b).

## Beyond Object-oriented

Imperative programming views everything as object with mutable states; in comparison, Clojure sees the world in terms of time, state and identity. The OO in a traditional sense has traits of imperative programming baked in, contrary to Clojure's philosophy (Fogus & Houser, 2014). Clojure has a "broad support" for various paradigms ; it can implement useful aspects in OO with its functional focus and immutability features (Fogus & Houser, 2014).

1. Polymorphism makes the system extensible and flexible. Clojure's `multimethods` separates polymorphism from OO and types (Hickey, 2015c). Instead of having a new data type for a new situation, Clojure would rather use a large collection of functions operating on a small set of data types. See the multimethods page on (Hickey, 2009).

2. Abstraction-oriented: Clojure provides the option of *ad hoc* hierarchy; this form of inheritance relationship can be defined among data types and even symbols. The Clojure equivalent of interface is called `protocol`.

3. Encapsulation hides the implementation details and enhances levels of abstraction. Clojure accomplishes encapsulation by immutability, closures and name spaces (Halloway, 2015b).

## Concurrent programming

Concurrency is easy to implement with Clojure. Since Clojure runs on JVM, it uses JVM threads. Clojure has language-level support for safe and lock-free concurrency (Rathore, 2011). Its basic tools for concurrency include `future`, `delay` and `promise`, which are all macros built in `clojure.core` (Higginbotham, 2015).

Clojure provides four mechanisms that ensure controlled state changes:

1. Macros such as ref, set, dosync expose Clojure's software transaction memory (STM). This feature supports sharing changing states synchronously, in coordination.

2. The Agent system also provide access to mutable states. Compared to the ref's, Agents provide independent and asynchronous access to data residing at individual locations.

3. Atoms share changing states in an independent and asynchronous manner between threads.

4. A Var "refer to a mutable storage location that can be dynamically rebound (to a new storage location) on a per-thread basis.", according Clojure's official website.

# Phase 3: Primitive Types

## What primitive types are available?

(integer? boolean?) What kinds of values can they have? (0? -0? true? -00.30?) What range of values can they have? (0 to infinity? -231 to 231?) How much memory does it each type use? (Bits and bytes - not all languages will let you know this. Some languages, like Python, make the integer grow to fit your needs.). Write a program to demonstrate as much of this as possible. Give a (short) explanation in comments.

### Scalar literal

Scalar literal is the closest thing to the familiar primitives. Since Clojure code runs on JVM, its data types are Java data types.

Clojure (Lisp, really) has the philosophy of code-as-data, meaning that its code is literally its own data structures and atomic values (Emerick, Carper, & Grand, 2012). The *reader* is used to interpret these data structure out of textual code; there are "noncollection values", or scalar literals, in the reader syntax. Some of the scalar literals do not have corresponding concepts in languages like Java or python.

Clojure's scalar literals include:

- Numbers
- Characters
- Strings
- Booleans
- `nil`
- Keywords
- Symbols
- Regular expressions

### Numbers

Clojure by defaults stores integers as Long, and floating point numbers as Double. Clojure can intelligently uses `java.lang".BigInteger` and `java.lang.BigDecimal` so that the maximum degree of precision is preserved; there is no limit to how big (or small) numbers can be in Clojure (Fogus & Houser, 2011). Consequently, there is no bound to the size of integers and floating point numbers.

### Rationals

A rational is a number with a nominator and a denominator, both of type arbitrary-precision integers. Rationals retain more precision than floating point numbers (Fogus & Houser, 2011).

```
;; Integers
user> (class 42) ;; get the class of literal 42
java.lang.Long
user> Long/SIZE  ;; get static member SIZE of class
64               ;; java.lang.Long, signed int
                 ;; 64 bits or 8 bytes

;; integer of arbitrary precision
;; N to force to use BigInt
user> (class 42N)
clojure.lang.BigInt

;; Floating points
user> (class 42.0)
java.lang.Double
user> Double/SIZE
64               ;; Both Long and Double are 64 bits
                 ;; IEEE floating point
                 ;; 64 bits or 8 bytyes

;; Floating point of arbitrary precision
;; M to force to use BigDecimal
user> (class 42M)
java.math.BigDecimal

;; Rational
user> (class 2/42)
clojure.lang.Ratio
```

### Special cases of numbers

Clojure deals with special cases without surprises.

```
;; comparing 0 to -0
user> (= 0 -0)
true

;; leading 0's are ignored
user> (class -00.30)
java.lang.Double
```

To run: in Emacs, `M-x`, select cider-jack-in Then do `C-c C-k` to compile the current buffer For using cider in Emacs, see

### Characters

Clojure character literals are denoted by a backslash prefix (Fogus & Houser, 2014). They are stored as Java characters, and Unicode characters are included, too. Since the size of a char is 16 bits, the range is from `\u0000` to `\uffff` (Oracle, 2015).

```
;; Of class java.lang.Character
clojure-noob.core> (class \u0042)
java.lang.Character
;; Unicode character
clojure-noob.core> \u0042
\B
clojure-noob.core> Character/SIZE
16  ;; 16 bits, 2 bytes
```

From (Fogus & Houser, 2014).

### String

Clojure strings are java strings. See comments in the code example for caveats in REPL environment.

```
(defn new-line
  []
  ;; A string is contained between two double quotes.
  ;; Including newlines.
  (println "I am a
oops... string...\n \toops"))
```

Output:

```
;; Run the new-line function
clojure-noob.core> (new-line)
I am a
oops... string...  ;; there IS a newline
    oops
nil

;; But REPL escapes backslash newline literals
clojure-noob.core> (def a-string"Hi\nthere")
#'clojure-noob.core/a-string
```

```
clojure-noob.core> a-string
"Hi\nthere"     ;; see
;; But the string does print as expected
clojure-noob.core> (println a-string)
Hi
there
nil
```

From (Fogus & Houser, 2014).

### Boolean

Java's true and false; they are not capitalized, nor are they represented by integers. Interestingly, Oracle documentation suggests that although booleans represent 1 bit of information, its size is not precisely defined (Oracle, 2015).

```
;; Cannot cast true into Integer
clojure-noob.core> (cast Integer true)
ClassCastException Cannot cast java.lang.Boolean
to java.lang.Integer java.lang.Class.cast
(Class.java:3369)
```

From (Rathore, 2011).

### nil

The famous null in Java. Logically equivalent to false. Some properties are shown below in the code example.

```
;; Does not equate to false
clojure-noob.core> (= nil false)
false
;; But logically is
clojure-noob.core> (if nil () (println "False!"))
False!  ;; else statement prints False!
nil     ;; every form evaluates to something
```

From (Rathore, 2011).

### Symbol

Symbols are objects that are often evaluated into something else (Fogus & Houser, 2014).

```
;; define a symbol called a-string
clojure-noob.core> (def a-string"Hi\nthere")
#'clojure-noob.core/a-string
;; evaluate that symbol, returns
;; a string, instead of the symbol itself
clojure-noob.core> a-string
"Hi\nthere"
;; But the evaluation can be suppressed
clojure-noob.core> `a-string ;; by `
clojure-noob.core/a-string
```

Adapted from (Fogus & Houser, 2014).

### Keyword

A keyword is similar to symbol, but a keyword evaluates to itself. Keywords are fare more commonly used than symbols (Fogus & Houser, 2014).

```
;; a keyword called mass-wasting
clojure-noob.core> :mass-wasting
;; evaluates to itself
:mass-wasting
;; usage as map keys
;; rocks symbol evaluates to a map, constructed by {}
;; ls is the key
clojure-noob.core> (def rocks {:ls "limestone"})
#'clojure-noob.core/rocks
clojure-noob.core> (get rocks :ls)
;; limestone is the value
"limestone"
```

Adapted from (Fogus & Houser, 2014).

### Regular expressions

In Clojure, a regular expressions is a string prefixed with a hashtag (Rathore, 2011). Although the usage of regular expression is far beyond the scope of this humble composition, 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
```

Inspired by (Fogus & Houser, 2014).

# Phase 4: Composite Types

## What composite types can you make?

(tuple, struct, matrix. . . ) List 3 or more. What formal definitions of constructors from your book and lecture do they each match? (record, cartesian product, array, union . . . ) If the type constructor has options, what choices did your language make? Give an example program using each of these type constructors?

## Answers

Collections are very important Clojure composite types. Clojure collections are all immutable. The elements in some collections can be accessed via keys, but no values can be set by keys. Clojure has four basic collection types:

- Lists
- Vectors
- Maps
- Sets

## List

Fun fact: the name Lisp comes from list processing (Fogus & Houser, 2014).

Clojure's list is a kind of mapping; it is also a recursive type (being a singly linked list). It is possible to access an item on the n-th index through `nth`; that is not the intended use of Clojure lists, because Clojure will do an O(n) traversal from the head (Fogus & Houser, 2014). Instead, lists are mainly used for denoting forms in Clojure (Fogus & Houser, 2014). The head of the list is resolved into "a function, macro or special operator", and the rest of the elements are evaluated in order, as parameters (Fogus & Houser, 2014).

Since collections in Clojure are immutable, the size of the list is fixed, and there is no limit in length when created (Fogus & Houser, 2014). A list can hold elements of any type (heterogeneous), including other collections. Operations are usually done to the head of a list, such as `conj`.

```
;; Constructed by ()'s
;; Yes the famous (((((())))))'s
;; (if...) is a list, as an element in (def....) list
;; So lists can contain objects of different types
;; Even other collections
clojure-noob.core> (def eg (if {} :truethy))
```

```
;; index from 0 to n-1, inclusive
;; Range is defined on non-negative integers
;; Else: IndexOutOfBoundsException from java
;; or a specified handler
clojure-noob.core> (nth (if nil "a") 1)
nil

;; Adding to the head of the list
;; put c in front of (a,b)
;; ' symbol suppresses evaluation
;; of list ("a" "b"), else error will occur
clojure-noob.core> (conj '("a" "b") "c")
;; the expr (conj ...) evaluates to a new list
;; the old list is still immutable
("c" "a" "b")

;; Empty test
user=> (empty? ())
true

;; get length
;; Need to suppress evaluation of the list
user=> (count '(1 42))
2
```

See (Fogus & Houser, 2014) and (dpritchett, 2015).

## Vector

Clojure vectors belong to mapping, since there is a correspondence from type integer (indices) to vector elements, which can assume any type (heterogeneous) (Fogus & Houser, 2014). The length of a vector is fixed once made; but there's no limit in length. An empty vector is not nil.

### Tuples in Clojure

Tuples are commonly built with vectors in Clojure (Rathore, 2011).

```
;; Constructed by [] `(literal syntax)
;; as in [:a "b" 42]: different types
;; multi-dimension is also possible
;; index from 0 to n-1, inclusive
;; Range is defined on non-negative integers
;; Else: IndexOutOfBoundsException from java
```

```
;; or a specified handler `(the (print.. ))
;; nth applies to vectors also
user=> (nth [:a "b" 42] 3 (println "bad index"))
bad index   ;; the handler prints
nil         ;; println's return value

;; alternative constructors
;; vector's arguments have an arbitrary length
clojure-noob.core> (vector 42 "42")
[42 "42"]
clojure-noob.core> (vec (range 2))
[0 1]       ;; vec takes a sequence

;; Example operation: appending an element
;; The expression (conj... ) evaluates
;;     to a new, longer vector
;; But the old vector itself is still immutable.
user=> (conj [:a :b] :c)
[:a :b :c]

;; Empty test
user=> (empty? [])
true

;; get length
user=> (count [])
0
```

See (Rathore, 2011) and (Fogus & Houser, 2014).

## Maps

Map is, by name, a kind of mapping. An element in a map consists of a key-value pair (Fogus & Houser, 2014). Clojure's map can have any type for both keys and values (heterogeneous) (Fogus & Houser, 2014). Since collections in Clojure are immutable, the size of the map is fixed throughout its lifetime. An empty map is not the same as nil (Fogus & Houser, 2014).

The keys in a map should not duplicate (Rathore, 2011). A map can be sorted or unsorted (hashmap) (Rathore, 2011).

```
;; {} constucts a map
;; with key-value pairs
;; {:answer 42 :who "DeepThought"} is the map
;; Recall :answer, :who are keywords
```

```
clojure-noob.core> (def ultimap {:answer 42 :who "DeepThought"})
#'clojure-noob.core/ultimap
;; Access a value through a key
clojure-noob.core> (get ultimap :answer)
42

;; Different types of keys and values are possible
;; Can have multidimensional maps
;; But the popular keys are still keyword type
;; note "a" and the first 42 is a pair;
;; for readability, the expression can also be
;; (def rand-map {"a" 42, 42 []})
clojure-noob.core> (def rand-map {"a" 42 42 []})
#'clojure-noob.core/rand-map

;; Get all the keys as a list
clojure-noob.core> (keys rand-map)
("a" 42)

;; Get all the values as a list
clojure-noob.core> (vals rand-map)
(42 [])

;; Empty test
user=> (empty? {})
true

;; Length is now the number of pairs
clojure-noob.core> (count rand-map)
2   ;; not 4

;; Sidenote of Clojure's truthiness
;; if empty map, then return :truthy
clojure-noob.core> (if {} :truthy)
:truethy    ;; {} looks to if like true
```

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

## Cartesian Products

### Sets

Clojure's set is a collection of objects without duplication (Rathore, 2011). Since a set can contain any data type (heterogeneous), it is a Cartesian product. It can come in a sorted or unsorted flavour (Rathore, 2011). There is no obvious

way to get set elements by name, because they technically don't have any, and nor is there a limit to the number of elements created in the set.

```
;; Constructed by #{} (literal)
user=> (def rocks #{:gneiss :slate :slate})
;; Duplicated elements are not allowed
IllegalArgumentException Duplicate key: :slate

;; Making a set from other collections
user=> (set [42 42 42])
#{42}   ;; duplicates eliminated

user=> (def rocks #{:gneiss :slate :schist})
#'clojure-noob.core/rocks       ;; OK

;; Empty test
user=> (empty? #{})
true

;; There seems to be no name access
;; contains? checks if the given key is present
user=> (contains? rocks 0)
false
user=> (contains? rocks :slate)
true

;; get length
user=> (count rocks)
3
```

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

## Interoperability with Java classes

Clojure can use Java classes (Fogus & Houser, 2014), a feature called *interoperability*. Clojure can access Java classes and their members, and even create new instances. Clojure provides mechanisms like `reify` and `deftype` to emulate Java interfaces (Fogus & Houser, 2014).

A class contains objects of different types; it is a Cartesian product.

```
;; Create an instance of Point class
;; The dot behind Point denotes creation
;; of an instance
clojure-noob.core> (java.awt.Point. 1 1)
#object[java.awt.Point 0x37a2bc27 "java.awt.Point[x=1,y=1]"]
```

Adapted from (Fogus & Houser, 2014).

## 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, 2015a). 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.

```
;; 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:

```
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.

```
(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 et al., 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)
```

## 20. Can a user define types (beyond the given type constructors)? If so, how? (typedef or #define in C)

There are three main type features in Clojure (see clojure.org/datatypes):

- `deftype`
- `defrecord`
- `reify`

Both `deftype` and `defrecord` generate *named* classes dynamically.

### Example usage of deftype:

```
;; defining the type
;; constructor takes two parameters, name & age
(deftype Animal [name age])

;; check: a java class indeed
user> (class Animal)
java.lang.Class

;; Usage

(deftype Animal [name age])
(def dog (Animal. "Worf" 10))
```

Output:

```
user> (.name dog)
"Worf"  ;; success
```

## 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?

### Math operations

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.

### Math on characters?

Clojure does not do math on characters. The closest thing is string concatenation.

```
usr> (str "one" " and another")
"one and another"   ;; success
```

### 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).

## 28. Can you do multi-dimensional arrays?

How? Are there limits? C and Java both allow this. Java even has some support for it.

### Answer: yes!

Clojure even has a function to build multi-dimensional arrays with specified dimension and length. There seems to be no but memory's limit to its maximum dimension.

```
user> (pprint (make-array Integer/TYPE 4 2))
[[0, 0], [0, 0], [0, 0], [0, 0]]
```

From clojuredoc (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.... ;; to whom it belongs
nil
```

### More about `if`

The argument in case of `else` condition is optional. Since Clojure has all expressions, `if` always returns something. If nothing is supplied to the `else` condition, Clojure defaults to return nil.

```
;; 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
    ;
```

## 41. Does it differentiate between/allow use of statements or expressions?

Clojure only uses expressions; no statement exists in Clojure (see clojure official site).

## 43. Does it use infix, prefix, postfix, mixfix operators? function calls? some combo of them?

### Answer

Prefix always. All function calls, unary, binary operators etc., are all prefix (Fogus & Houser, 2014).

## 45. What control structures does it use? For instance C has an old style ?: for selection.

Clojure has two types of flow control: normal and exceptional (see here).

- Normal types of control structures include the usual if, if-let, when, when-let, cond, do, eval, loop and more.
- Exceptional circumstances include assert and try-catch.

## if

```
;; if
user=> (if nil "not" "is nil")
"is nil"
```

## when

```
;; An example for when
(defn train ;; define a function, train
    [x]     ;; that takes one argument
    (when (pos? x)  ;; when x is positive
      (println "Choo choo!")
      ;; recur is a special form that does
      ;; __tail recursion__
      (recur (dec x))))
```

Lovely output:

```
clojure-noob.core> (train 3)
Choo choo!
Choo choo!
Choo choo!
nil
```

## when-let

`when-let` is a convenient way to execute *when* something is not `nil`.

```
;; when-let is useful for dealing with empty collection
(defn get-head
  [coll]
  (when-let [handle (seq coll)]
    (first handle)))
```

Output:

```
user=> (get-head [1,43,34])
1
user=> (get-head [])
nil
```

See clojuredoc.

# 46. Language pre-processors and macros

## Answer: yes Clojure have them

The *times* in Clojure (Fogus & Houser, 2014):

- read time
- *macro-expansion time*
- compile time
- run-time

Clojure's macro-expansion can be viewed as pre-processing (Kingsbury, 2015). During this phase prior to compilation, the Clojure code is restructured.

```
;; use macroexpand to see what's under the hood
user> (macroexpand `(-> 25 Math/sqrt list))

;; expanded version
(clojure.core/list (java.lang.Math/sqrt 25))

;; the expressions evaluate to 5 as expected
user> (-> 25 Math/sqrt list)
(5.0)
user> (clojure.core/list (java.lang.Math/sqrt 25))
(5.0)
```

## Macros compared: Clojure and C

The C preprocessor uses its own syntax (not C) and operates *on text*. Since Lisp / Clojure has the philosophy of "data is code is data", macro-expansion is written in Clojure and happens on data structures. In fact, many more fundamental functions in Clojure are written as macros, and users can rewrite them or define their own macros (Kingsbury, 2015).

```
;; source code for operator and
user> (source and)
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
nil

;; overwriting the macro
user> (defmacro and
  ([x y] (println "oops")))
#'user/and
user> (and 1 2)
oops
nil
```

## 47. Does it claim to be portable like Java? How portable is it?

Rich Hickey, the designer of Clojure, said that Clojure was intended as a hosted language . Clojure can use Java libraries easily. This hosted feature allows Clojure be used wherever Java is used . Consequently, Clojure is very portable because JVM is; portability is one of the key features of Clojure. Besides JVM, Clojure can also be hosted on CLR (Common Language Runtime) and JavaScript.

See this interview with Hickey.

## 52. What comment stypes does it support?

Clojure has two ways of commenting:

1. commenting out a line with `;`
2. commenting out a form with `#_`

```
; This is a comment
;; but using two of ;'s is more common
user=> (if true (:answer) (+ 1 41))
IllegalArgumentException Wrong number of args
;; #_ comments out a pair of ()
```

```
user> (if true #_ (:answer) (+ 1 41))
42
;; #_ works for [] as well
user> (if true #_ [0] [42])
[42]
```

See this Clojure style guide @ github.

## 99. Partially apply constructor of a class?

Since Haskell and Ocaml take different design decisions about this...

### Answer: yes

Yes, we can use the higher order function `partial`.

Define a class:

```
;; constructor takes two parameters, name & age
(deftype Animal [name age])
```

Applying the constructor:

```
;; check: a java class indeed
user> (class Animal)
java.lang.Class

;; partial returns a function short of one parameter
user> (map (partial #(Animal. "a name" %)) [1 2 3])
;; mapped the constuctor and created a lot of animals
(#object[user.Animal 0x52845f1 "user.Animal@52845f1"]
 #object[user.Animal 0x5857a1a1 "user.Animal@5857a1a1"]
 #object[user.Animal 0x4ec5bfca "user.Animal@4ec5bfca"])
```

## References


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. (2011). *The joy of clojure: Thinking the clojure way.*

Manning Publications Co.

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

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

Halloway, S. (2015b). Rifle-oriented programming with clojure. *Accessed Online from Http:// Thinkrelevance.com /Blog/2009/08/12/rifle-Oriented-Programming-with-Clojure-2*.

Hickey, R. (2009). Clojure homepage. *Accessed Online from: Http://www. Clojure. Org on, 12*.

Hickey, R. (2015a). Clojure - dynamic. *Accessed Online from: Http://www. Clojure. Org / Dynamic on*.

Hickey, R. (2015b). Clojure - functional_programming. *Accessed Online from: Http://www. Clojure. Org / Functional_programming on*.

Hickey, R. (2015c). Clojure - rationale. *Accessed Online from: Http://www. Clojure. Org / Rationale on*.

Higginbotham, D. (2015). Concurrency, parallelism and states. and zombies. | clojure for the brave and true. *Accessd Online from Http:// Www.braveclojure.com/ Concurrency/*.

Kingsbury, K. (2015). Clojure from the ground up: Macros. *Https://aphyr.com/posts/305-Clojure-from-the-Ground-up-Macros*.

Oracle. (2015). Primitive data types. *Accessed Online from Https:// Docs.oracle.com/ Javase/ Tutorial/ Java/ Nutsandbolts/ Datatypes.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*.