(Clojure (for the masses))

(author "Tero Kadenius" "Tarmo Aidantausta")

(date "30.03.2011")

Contents

1 Introduction

This seminar paper introduces Clojure, a dialect of Lisp, which is a dynamically typed functional programming language hosted on virtual machines like JVM and CLR.

The key point that makes Clojure stand out in the pool of new and fancy programming languages is the fact it has been developed from the ground up with multithreaded programming in mind. *Immutable data structures*, support for *software transactional memory* (STM) with the combination of *functional programming* paradigm give Clojure good tools working with concurrency. [53]

Even though concurrency had an emphasis in designing Clojure it is still a general purpose language with interoperability features to leverage the functionality and the ecosystem of the host platform, so it's possible to reuse already existing code and libraries done for the host platform. [46, 44]

The paper gives first a short tour to very basics of Clojure and dives into further details of the language and its features.

1.1 Dialect of Lisp

```
(= 2 (+ 1 1))
```

> "Lisp is a family of computer programming languages based on formal functional calculus. Lisp (for "List Processing Language") stores and manipulates programs in the same manner as any other data, making it well suited for "meta-programming" applications." [1]
>
> "Lisp has jokingly been called "the most intelligent way to misuse a computer". I think that description is a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts." [27]

1.2 Dynamic typing

> "In a *dynamically typed* language, values have fixed types, but variables and expressions have no fixed types." [59]

1.3 Functional programming

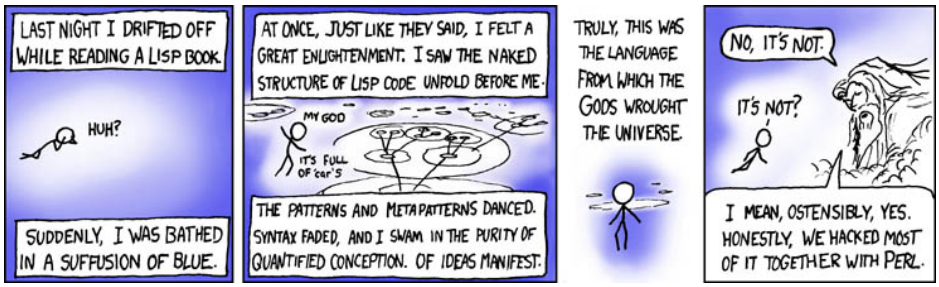
> "In functional programming, the model of computation is the application of functions to arguments." [59]

2 Introduction to Clojure syntax

To make it easier to read this paper, parts of the syntax of Clojure are introduced here with simple shortly explained examples. There are a lot of concepts that

are not explained but will be explained later in the article.

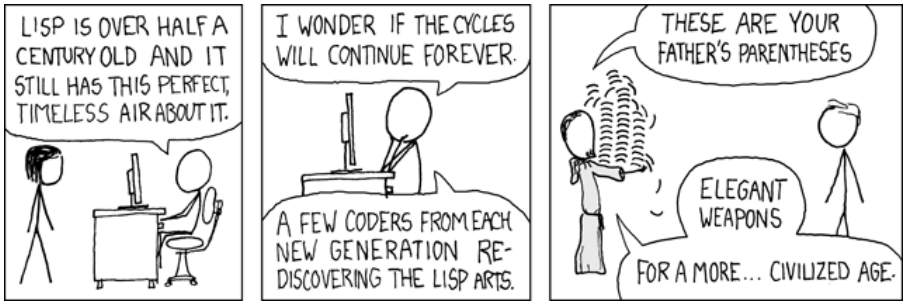
2.1 Lispy syntax



> "At once, just like they said, I felt a great enlightement. I saw the naked structure of Lisp code unfold before me." [55]

Lisps, like Clojure, don't have a lot of syntax in the traditional sense of syntax, compared to languages like Java, C or C++. The syntax is minimized to mainly into defining lists with parenthesis and the different rules of evaluations of those lists.

2.1.1 Parentheses, parentheses, parentheses



> "These are your father's parentheses. Elegant weapons, for a more... civilized age." [56]

As Lisps are about lists - the name LISP derives from "LISt Processing" [54], and lists in Lisps are about parentheses. This means that, in Clojure, reading and writing parentheses is inevitable. A lot has been said about the amount of parentheses in Lisps, both good and bad.

> "Lisp has all the visual appeal of oatmeal with fingernail clippings mixed in." [58]
>
> "...and here I thought it was LotsofInfernalStupidParentheses. My mistake; I must have just been in a worse mood. ;->" [57]

2.1.2 Lists

In Clojure, as in all the Lisps, the lists have special syntax to them which is due to the homoiconicity - code is data and data is code. This might seem confusing at first but the rules are quite simple although they overload the definition of lists a bit.

Below we can see a list which evaluates to a function call to + with 1, 2, and 3 as parameters to that function.

```
(+ 1 2 3)
```

Below is a definition of list with numbers. Quote in the beginning of the line tells the reader that it should treat the following list just as data.

```
'(+ 1 2 3)
```

Below we can see how a list can be also constructed with a function call to a list.

```
(list + 1 2 3)
```

2.1.3 Prefix vs. infix notation

> "Polish notation, also known as prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators to the left of their operands."[21]

```
+ 1 2
```

> "Infix notation is the common arithmetic and logical formula notation, in which operators are written infix-style between the operands they act on (e.g. 2 + 2)." [15]

```
1 + 2
```

Clojure, as all Lisps, uses prefix notation in contrast to infix notation is which used in languages like C, C++ and Java. [21, 15]

2.1.4 Defining functions

Functions are first-class objects in Clojure and there is more than one way of defining them.[43]

```
(def hai (fn [] "Ou hai!"))
```

Above we define a function that returns string "Ou hai!" by using *macros* called *fn* and *def*. *fn* that creates the function takes names and the default values for parameters inside [] as the first parameter and the body of the function as the second parameter.[36]A macro called *def* binds a the name with a value.[36]

You can also define functions with a *macro* called *defn*. [43]

```
(defn hai-u [u]
  (str "Hai, " u))
```

That *macro* takes the name, optionally a document string and attribute map, parameters and the function body as parameters.[36]

You can also use a Clojures dispatch macro to create a function. [47]

```
(def hai-u2 #(str "Hai, " %1 " and " %2))
```

3 Functional vs. imperative programming

3.1 Imperative programming

> "Imperative programming is a programming paradigm that describes computation in terms of statements that change a program state."[14]

Nearly all machine code implementations are written in imperative style. The contents of the memory holds the state and machine language instructions modify it. Higher-level imperative languages have more advanced features like variables and complex statements, but the basic idea remains the same. [14]

Here is a small snippet of imperative code. It has a notion of state (a), which is mutated. In addition, an IO operation is performed.

```
int a = 3;
int b = 4;
if (a < b) {
        a++;
}
print(a);
```

3.2 Object oriented programming

> "The set of values of the attributes of a particular object is called its state. The object consists of state and the behavior that's defined in the object's classes." [19]

Object oriented programming provides a feature called encapsulation. Encapsulation prevents users of the object from direcly modifying the data that forms the state by providing operations (methods) for doing it. This is done in order to ensure the validity of the internal state of the object. [8]

In other words, at its heart, object oriented programming tends to be imperative. The paradigm itself doesn't enforce it, but that is usually the case. [50]

An example that demonstrates the imperative nature of OO code:

```
class Foo {
    int a = 3;
        int b = 4;

    increment() {
        if (a < b) {
                    a++;
           }
        }

        print() {
                print(a);
        }
        ....
}
```

What happens here is identical to the imperative code example except that in this case the data (a and b) and the operations mutating the state (a++) or causing other side effects (print(a)) are encapsulated inside the object.

increment() is an instruction for modifying the state of the object. The result of increment() may vary in different points of time depending on the state of the object.

3.3 Functional programming

> "Functional programming has its roots in mathematics. Instead of providing instructions for modifying the state of the program, functional programming emphasizes the application of functions and avoid state and mutable data in general. "[11]

3.3.1 Functions as first class objects

The notion of a function is not unique to functional programming languages. However, functional languages have what is called *first class functions*. This means that functions have a central role in the code, much like objects do in OO languages. Functions can be stored to data structures and the use of *higher-order functions* is common. [9] The objective of having no side effects manifests itself in *pure functions*.

3.3.2 Pure functions

Function is considered pure if:

1. "The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change as program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices. [4]

2. "Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices." [4]

Using pure functions has several benefits:

1. Pure expression can be removed without affecting other expressions if the result of the pure expression is not used. [11]
2. *Referential transparency.* An expression can be replaced with its value without causing changes to the program. The output is always the same with the same input. [23]
3. If a pure function does not depend on the result of another pure function, they can be performed in any order. Ie. they are thread-safe and can be run in parallel without typical concurrency issues. [11]
4. Lack of side effects guaranteed by the language, provides opportunities for compiler optimizations. [11]

3.3.3 Higher-order functions

Higher-order function is a function that either takes one or more functions as parameters or returns one as a value. [13] Well-known examples of higher-order functions are *map* and *fold [10].* "Map is the name of a higher-order function that applies a given function element-wise to a list of elements and returns a list of results." [16]. Fold is a "function that iterate an arbitrary function over a data structure in some order and build up a return value". [10]

- Doubling the value of every element in a list using map:

  ```
  (map (fn [x] (* 2 x)) '(1 2 3))
  ```

  => ((2 1) (* 2 2) (* 2 3)) => (2 4 6)*
  *infix notation equivalent: (2 * 1) (2 * 2)(2 * 3)*

- In Clojure fold is called reduce. A trivial example for calculating the sum of the elements in a list:

  ```
  (reduce + '(1 2 3))
  ```

  => (+ (+1 2) 3) => 6
  infix notation equivalent: (1+2) + 3

Partial function application and currying

Higher-order functions enable an interesting feature where a new function can be generated based on another function. *Partial function application* is a technique which produces a function in which one or more of the arguments of the original function are fixed. *Currying* resembles partial function application. The difference is that in currying each (curried) function takes only a single argument and

produces a function which takes one argument less than its predecessor. Ie. currying produces a chain of functions, whereas with partial function application arbitrary number of functions can be fixed at once. [7]

Out of the box, Clojure supports partial function application but not currying.

A simple example where a function that adds 2 to its argument is applied to a list of elements:

- `(map(partial + 2) '(1 2 3))`

  => ((+ 2 1)(+ 2 2)(+ 3 3))=> (3 5 7)
  infix notation equivalent: ((2 + 1)(2 + 2)(2 + 3))

3.4 Differences

Characteristic	Imperative approach	Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

[12]

3.5 Critique

The proponents of functional programming claim that imperative programming is fundamentally broken - especially in a multi-threaded environment. First of all, there is an argument that the world doesn't function in a way imperative programming models it. When dealing with mutable state, the "world" has to stop in order it be examined or changed. This becomes a major problem when bringing concurrent programming to the picture. [50]

OO programming suffers from the same problems as imperative programming. To quote Rich Hickey, the creator of Clojure: "Encapsulation just means: I'm in charge of this spaghetti code." Ie. encapsulation doesn't change the fact that OO is usually based on mutable state. It just tries to prevent the user of object's interface from seeing it (the (imperative) spaghetti code).

4 Closer look at Clojure

Now that the paradigm of functional programming has been introduced, some of the details of Clojures features and terminology is explained.

4.1 Syntax

> "Clojure is a homoiconic language, which is a fancy term describing the fact that Clojure programs are represented by Clojure data structures." [47]

Clojure syntax is built on *symbolic expression*s, *S-expression*s, that are list based data structures. [24, 47, 5] In addition to lists also symbols, literals, vectors, maps and sets make up the syntax and are parsed by the Clojure reader. [47]

4.1.1 Reader

The reader parses the textual presentation of the Clojure code to data structures.

```
(doc read)
```

Documentation for any given function can be acquired by calling the function doc.

Then it creates the form of that same data structure that the compiler will see. Clojure compiler compiles the code, data structures, to host platform bytecode. This bytecode is then executed by the host platform virtual machine. [52]

4.1.2 Symbols

> "Symbols begin with a non-numeric character and can contain alphanumeric characters and *, +, !, -, _, and ?." [47]

```
(def ines "a symbol called ines")
```

def is a macro that takes a symbol as a parameter and then gives that symbol a value if one is given. Here it defines a symbol called ines in the current namespace with the value "a symbol called ines". [36]

4.1.3 Literals

```
'((strings are literals as are numbers and characters  (1 2 3) (\a \b \c))
  ( and of course booleans and and keywords true :keyword))
```

4.1.4 Lists

```
(list "can contain anything you want" 1 2 3 \a \b \c :keyword '())
```

If you don't use ' or quote Clojure will try to use the first cell off a list as a function call. So if you want to just express data, use ' or quote.

```
'(any of these won't be evaluated)
```

4.1.5 Vectors

```
(vector '[can contain anything too 1 2 3 \a \b \c :keyword])
```

4.1.6 Maps

```
(hash-map :key value key :value :map {:can contain :maps too})
```

4.1.7 Sets

```
(hash-set
  "can contain sets" #{:a :b} #{:b :c}
  "and anything unique" 1 2 3 \a \b \c '(:a :b))
```

4.2 Macros

> "The definition of a macro is essentially a function that generates Lisp code—a program that writes programs."
>
> "Macros work differently from normal functions, and knowing how and why macros are different is the key to using them correctly. A function produces results, but a macro produces expressions—which, when evaluated, produce results." [29]

In Clojure macros are implemented in a way that the compiler can be extended by user code - you can really grow the language. [45]

```
(defmacro print-evaluate
  [code]
  `(println '~code " evaluates to " ~code))
```

> *"defmacro* defines a new macro with similar structure to *defn*. The ` is used to create a template expression, where we can evaluate certain items within the expression by using macro characters (#,~,',list-frag?). " [28]

- defmacro macro takes the name (symbol), parameter (vector) and the body (expression) as parameters [34]
  - print-evaluate
  - [code]
  - `(println '~code " evaluates to " ~code)
- ` , which is called either *tick* or *backquote,* stops evaluation [47]
- '~code is equivalent to (quote ~code) [47]

- quote stops evaluation [47]
- ~code unquotes the ‘ [47]

This very simple example only scratches the surface what you can do with macros, but it demonstrates at least one way of creating them.

4.3 Evaluation

> "Every form not handled specially by a special form or macro is considered by the compiler to be an expression, which is evaluated to yield a value. There are no declarations or statements, although sometimes expressions may be evaluated for their side-effects and their values ignored."

```
(println is a symbol that is bound to a function value)
```

Clojure code can be evaluated interactively with REPL, forms read from a stream via load or load-file or programatically with *eval*. [42]

> "In all cases, evaluation is the same - a single object is considered by the compiler, evaluated, and its result returned. If an expression needs to be compiled, it will be." [42]

4.4 Read-Eval-Print-Loop

> "A read-eval-print loop (REPL), also known as an interactive toplevel, is a simple, interactive computer programming environment." [22]

Clojure has a REPL which you can use to interact with your code - you can grow your program, with data loaded, adding features, fixing bugs, testing, in an unbroken stream. [41]

4.5 Data structures

The data structures (collections) in Clojure are persistent [50, 3, 39]. The implementation of Clojure collections allows efficient (semantic) copying. The efficiency is achieved by utilizing structural sharing. [39] This is possible due to the immutability. Ie. structural sharing between mutable data structures would be problematic.

Collections are not bound to concrete data structures. This is a big difference between Clojure and some older Lisps. [40]Instead, in Clojure, collections are represented by abstractions. Each abstraction may have one or more implementations. "Clojure's reader supports literal syntax for maps, sets and vectors in addition to lists." [5] The The literal syntax for the aforementioned data structures was introduced in chapter 4.1.

4.5.1 Sequences

Another key thing related to data structure abstraction in Clojure is sequences (seqs). Many algorithms in Clojure use sequences as their data structure abstraction. "A seq is a logical list, and unlike most Lisps where the list is represented by a concrete, 2-slot structure" [48]Seq interface (ISeq) allows many data structures to expose their elements as sequences. "The seq function yields an implementation of ISeq appropriate to the collection. Seqs differ from iterators in that they are persistent and immutable, not stateful cursors into a collection. As such, they are useful for much more than foreach - functions can consume and produce seqs, they are thread safe, they can share structure etc."[48] Clojure provides an extensive library of functions for processing sequences. [48] All sequence functions can be used with any collection. [39]

4.6 Data types

http://clojure.org/datatypes

4.7 Control structures

if, *do*, and *loop/recur* are three most common control structures in Clojure. [30]

4.7.1 if

```
(println (if (= 42 (* 6 7)) "true" "false"))
```

4.7.2 do

```
(defn grade-good?
  [grade]
  (if (> grade 3)
    "Good it is! :)"
    (do
      (println "Low grade!" grade)
      "No, it's bad. :(")))
```

Do is idiomatic to use when you're introducing side-effects. Do ignores the return values all the other forms besides the last. [30]

4.7.3 loop/recur

> "The loop special form works like *let*, establishing bindings and then evaluating exprs. The difference is that *loop* sets a recursion point, which can then be targeted by the *recur* special form."
>
> "*recur* binds new values for *loop*'s bindings and returns control to the top of the *loop*." [30]
>
> "If the recursion point was a fn method, then it rebinds the params. If the recursion point was a loop, then it rebinds the loop bindings."[49]

```
(loop
  [x '(1 2 3 4) sumx 0]
  (if (empty? x)
    sumx
    (recur (rest x) (+ (first x) sumx))))
```

The code example above iterates through the list of numbers by reducing a number every recursion from the list x and adds it up to the sumx.

- x is set to (1 2 3 4) and sumx to 0.
- first returns the first item in the collection. [48]
- + returns the addition of two numbers. [34]
- rest returns a sequence of the items after the first. [48]

Loop/recur is a couple that main exists in Clojure due to deficiencies of tail-call-recursion on host platforms.

5 Concurrency in Clojure

5.1 From serial to parallel computing

For many years the speed of majority of computer programs could be improved by upgrading the hardware on which the program was run. Ie. "Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004." [20]Therefore, the standard practice has been to write software for serial computation meaning that: [26]

- Software is run on a single computer on a single processor (core)
- Only one instruction may execute at any moment in time.
- Instructions are executed one after another.

Moore's law states that the number of transistors that can be placed in an integrated circuit doubles approximately every two years. The trend started in 1958 and is expected to continue until 2015 or 2020 or later. [17] The extra transistors cannot be used for increasing the frequency of the microprocessor, but they can be used for adding new processor cores for parallel computing [20].

5.2 Problems caused by imperative programming paradigm

As stated earlier, by avoiding mutable state there is no need for locking and threads cannot interfere with each other. Let's take a look at two common concurrency issues.

- "*A deadlock* occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked." [2]

- *"Race condition* occurs when two threads access a shared variable at the same time." [2]

As we can see these issues are not necessarily related to the underlying problem we are trying to solve. They are mere implementation details.

5.3 Simple things should be simple

Immutable data structures and pure functions enable trivially easy parallel processing of functions as long as there is no data dependency between them.

A simple example of easy parallel execution of functions using pmap, a parallel map implementation:

```
(pmap (fn [x] (+ x 2)) '(1 2 3))
```

There is no need for explicitly spawning new threads nor is there fear for race conditions or deadlocks since the function does not rely on external state or mutate anything.

5.4 Reference types

"Clojure, being a practical language, allows state to change but provides mechanism to ensure that, when it does so, it remains consistent, while alleviating developers from having to avoid conflicts manually using locks etc."[38] In practice, this means that Clojure has extensive concurrency features built-in.

Clojure provides 4 different mechanisms for maintaining a persistent reference to a changing value. They are: [51]

1. Vars
2. Atoms
3. Agents
4. Refs

The reference types can be divided into 2 categories based on how they are modified. Vars, Atoms, and Refs are modified synchronously meaning that when a function is applied to a syncronous reference type, the call blocks untill the function has been applied. Agent is the only asynchronous reference type. [6, 51] Atom, Agent and Ref are created by calling a specific function, unsurprisingly *atom*, *agent* and *ref* respectively and the value each reference type is holding, can be accessed with the function *deref* or the reader macro *@*.

5.4.1 Vars

Vars resemble global variables in other programming languages. [wikib] Vars can have a binding to an initial value called a root binding. A root binding is shared by all threads unless the var has a per-thread binding. Therefore, the

value of a var is its per-thread binding or if it doesn't have any, its root binding. If neither binding exist, then the var is unbound. [51]

Vars are created using the special form *def [51, 49]*. Var is bound a given value, otherwise it is unbound. If a value was not supplied and the var did exist and had a value, the old value remains bound. [51] Rebinding the same var with def is not encouraged since "Subsequently calling (def something 6) is not a thread-safe operation." [6]

Repl examples:

```
user=> (def foo)
#'user/foo

user=> foo
java.lang.IllegalStateException: Var user/foo is unbound. (NO_SOURCE_FILE:0)

user=> (def foo 3)
#'user/foo

user=> foo
3

user=> (def foo)
#'user/foo

user=> foo
3

user=> (def foo 4)
#'user/foo

user=> foo
4
```

5.4.2 Atoms

"Atoms provide a way to manage shared, synchronous, independent state." [33]

"Shared" means that they can be modified from different threads.

"Synchronous" means that the call function modifying an atom blocks until the operation is performed.

"Independent" means that there is no coordinated mechanism for ensuring that only one thread at a time can modify the value of an atom. Atom uses a different technique for achieving the same goal. The value of an atom is changed by applying the function *swap!* to its old value. "swap! reads the current value, applies the function to it, and attempts to compare-and-set it in. Since another thread may have changed the value in the intervening time, it may have to retry, and does so in a spin loop. The net effect is that the value will always be the

result of the application of the supplied function to a current value, atomically. However, because the function might be called multiple times, it must be free of side effects." Atoms are created by calling the function *atom*. [33, 35, 37].

Repl examples:

```
user=> (def foo (atom 1))
#'user/foo

user=> (swap! foo inc)
2

user=>(let [bar (atom [:a])]
            (println @bar)
            (swap! bar conj :b))
[:a]
[:a :b]
```

5.4.3 Agents

Agents are the asynchronous counter-part of atoms. They are uncoordinated like atoms are, meaning that "atoms and agents queue up change functions to ensure that the changes occur atomically." [6] Agents are modified by applying the function *send* on them. Send applies (sends) a function to the agent, which is used for modifying the value of the agent. However, unlike the case of atom, the function sent is not applied to the current value immediately. Instead the call to send returns immediately. [6]

The state of the agent can be requested by adding a watcher to the agent. [32] Alternatively, a blocking function await can be called. await "blocks the current thread (indefinitely!) until all actions dispatched thus far, from this thread or agent, to the agent(s) have occurred."[34]

Repl examples:

```
user=> (def foo (agent 1))
#'user/foo

user=> @foo
1

user=> (do (send foo inc) (await foo))
nil

user=> @foo
2
```

5.4.4 Refs

Refs are mutable references bound to a single storage location. The key thing about Refs is that they are transactional. Ref modification has to happen within a coordinated transaction. [31] Enforcing the use of a transaction eliminates the possibility of a conflict when two threads update a Ref. [6]

Refs are created by calling the function *ref*. The value of a Ref is modified by using functions *ref-set*, *alter* or *commute*. A transaction is enabled by using the macro *dosync*.

Repl examples:

```
user=> (def foo (ref "bar"))
#'user/foo

user=> @foo
bar

user=> (ref-set foo "impossible")
java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)

user=> (dosync (ref-set foo "success"))
"success"

user=> @foo
"success"
```

5.5 Software transactional memory (STM)

What exactly does a transaction, as referred in the last section, mean? Clojure has a clever mechanism for automatic handling of transactions called *software transactional memory (STM) [38].*"(STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It is an alternative to lock-based synchronization. A transaction in this context is a piece of code that executes a series of reads and writes to shared memory. " *[25]* STM implementations have been written for a number of languages as some kind of an API or binding but in Clojure, STM is built directly into the language core. *[25]*

Clojure transactions are similar to those found in database management systems. The STM implemention in Clojure ensures that all actions on Refs are atomic, consistent and isolated. [31]

- "Atomic means that every change to Refs made within a transaction occurs or none do." [31]
- "Consistent means that each new value can be checked with a validator function before allowing the transaction to commit." [31]

- "Isolated means that no transaction sees the effects of any other transaction while it is running."[31]

If a transaction encounters a conflict while running, it is automatically retried. On a more defailed level the implementation guarantees that: [31]

1. "All reads of Refs will see a consistent snapshot of the 'Ref world' as of the starting point of the transaction (its 'read point'). The transaction will see any changes it has made. This is called the in-transaction-value."
2. "All changes made to Refs during a transaction (via ref-set, alter or commute) will appear to occur at a single point in the 'Ref world' timeline (its 'write point')."
3. "No changes will have been made by any other transactions to any Refs that have been ref-set/altered/ensured by this transaction."
4. "Changes may have been made by other transactions to any Refs that have been commuted by this transaction. That should be okay since the function applied by commute should be commutative."
5. "Readers and commuters will never block writers, commuters, or other readers."
6. "Writers will never block commuters, or readers."
7. "I/O and other activities with side-effects should be avoided in transactions, since transactions will be retried. The io! macro can be used to prevent the use of an impure function in a transaction."
8. "If a constraint on the validity of a value of a Ref that is being changed depends upon the simultaneous value of a Ref that is not being changed, that second Ref can be protected from modification by calling ensure. Refs 'ensured' this way will be protected (item #3), but don't change the world (item #2)."
9. "The Clojure MVCC [18]STM is designed to work with the persistent collections, and it is strongly recommended that you use the Clojure collections as the values of your Refs. Since all work done in an STM transaction is speculative, it is imperative that there be a low cost to making copies and modifications. Persistent collections have free copies (just use the original, it can't be changed), and 'modifications' share structure efficiently."
10. "The values placed in Refs must be, or be considered, immutable!"

References

[1] Lisp programming language. Available from: http://en.wikiquote.org/wiki/Lisp_programming_language.

[2] Description of race conditions and deadlocks. December 2006. Available from: http://support.microsoft.com/kb/317723.

[3] Persistent data structure. *Wikipedia*, 2010. Available from: http://en.wikipedia.org/wiki/Persistent_data_structure.

[4] Pure function. *Wikipedia*, 2010. Available from: http://en.wikipedia.org/wiki/Pure_function.

[5] Clojure. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Clojure.

[6] Clojure programming/by example. *Wikibooks*, March 2011. Available from: http://en.wikibooks.org/wiki/Clojure_Programming/By_Example.

[7] Currying. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Currying.

[8] Encapsulation (object-oriented programming). *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29.

[9] First-class function. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/First-class_function.

[10] Fold (higher-order function). *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Fold_%28higher-order_function%29.

[11] Functional programming. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Functional_programming.

[12] Functional programming vs. imperative programming. web, 2011.

[13] Higher-order function. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Higher-order_function.

[14] Imperative programming. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Imperative_programming.

[15] Infix notation. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Infix_notation.

[16] Map (higher-order function). *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Map_%28higher-order_function%29.

[17] Moore's law. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Moore%27s_Law.

[18] Multiversion concurrency control. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Multiversion_concurrency_control.

[19] Object-oriented programming. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Object-oriented_programming.

[20] Parallel computing. *Wikipedia*, March 2011. Available from: http://en.wikipedia.org/wiki/Parallel_computing.

[21] Polish notation. *Wikipedia*, 2011. Available from: http://simple.wikipedia.org/wiki/Prefix_notation.

[22] Read-eval-print loop. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Read-eval-print_loop.

[23] Referential transparency (computer science). *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Referential_transparency_%28computer_science%29.

[24] S-expression. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/S-expression.

[25] Software transactional memory. *Wikipedia*, 2011. Available from: http://en.wikipedia.org/wiki/Software_transactional_memory.

[26] B. Barney. Introduction to parallel computing. *Lawrence Livermore National Laboratory*. Available from: https://computing.llnl.gov/tutorials/parallel_comp.

[27] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15:859–866, October 1972. Available from: http://doi.acm.org/10.1145/355604.361591, doi:http://doi.acm.org/10.1145/355604.361591.

[28] J. Foster. Clojure macros. *fatvat.co.uk*, 2009. Available from: http://www.fatvat.co.uk/2009/02/clojure-macros.html.

[29] P. Graham. *On Lisp*. Prentice Hall, 1994.

[30] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[31] R. Hickey. Refs and transactions. *clojure.org*. Available from: http://clojure.org/refs.

[32] R. Hickey. Agents and asynchronous actions. *clojure.org*, 2010. Available from: http://clojure.org/agents.

[33] R. Hickey. Atoms. *clojure.org*, 2010. Available from: http://clojure.org/atoms.

[34] R. Hickey. Clojure v1.2 api documentation. *clojure.org*, 2010. Available from: http://clojure.github.com/clojure/.

[35] R. Hickey. clojure.core - atom. *clojure.org*, 2010. Available from: `http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/atom`.

[36] R. Hickey. clojure.core - clojure v1.2 api documentation. *clojure.org*, 2010. Available from: `http://clojure.github.com/clojure/clojure.core-api.html`.

[37] R. Hickey. clojure.core - deref. *clojure.org*, 2010. Available from: `http://clojure.github.com/clojure/clojure.core-api.html#clojure.core/deref`.

[38] R. Hickey. Concurrent programming. *clojure.org*, 2010. Available from: `http://clojure.org/concurrent_programming`.

[39] R. Hickey. Data structures. *clojure.org*, 2010. Available from: `http://clojure.org/data_structures`.

[40] R. Hickey. Differences with other lisps. *clojure.org*, 2010. Available from: `http://clojure.org/lisps`.

[41] R. Hickey. Dynamic development. *clojure.org*, 2010. Available from: `http://clojure.org/dynamic`.

[42] R. Hickey. Evalution. *clojure.org*, 2010. Available from: `http://clojure.org/evaluation`.

[43] R. Hickey. Functional programming. *clojure.org*, 2010. Available from: `http://clojure.org/functional_programming`.

[44] R. Hickey. Java interop. *clojure.org*, 2010. Available from: `http://clojure.org/java_interop`.

[45] R. Hickey. Macros. *clojure.org*, 2010. Available from: `http://clojure.org/macros`.

[46] R. Hickey. Rationale. *clojure.org*, 2010. Available from: `http://clojure.org/rationale`.

[47] R. Hickey. The reader. *clojure.org*, 2010. Available from: `http://clojure.org/reader`.

[48] R. Hickey. Sequences. *clojure.org*, 2010. Available from: `http://clojure.org/sequences`.

[49] R. Hickey. Special forms. *clojure.org*, 2010. Available from: `http://clojure.org/special_forms`.

[50] R. Hickey. Values and change - clojure's approach to identity and state. *clojure.org*, 2010. Available from: `http://clojure.org/state`.

[51] R. Hickey. Vars and the global environment. *clojure.org*, 2010. Available from: http://clojure.org/vars.

[52] R. Hickey. Ahead-of-time compilation and class generation. *clojure.org*, 2011. Available from: http://clojure.org/compilation.

[53] J. M. Kraus and H. A. Kestler. Multi-core parallelization in clojure: a case study. In *Proceedings of the 6th European Lisp Workshop*, ELW '09, pages 8–17, New York, NY, USA, 2009. ACM. Available from: http://doi.acm.org/10.1145/1562868.1562870, doi:http://doi.acm.org/10.1145/1562868.1562870.

[54] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3:184–195, April 1960. Available from: http://doi.acm.org/10.1145/367177.367199, doi:http://doi.acm.org/10.1145/367177.367199.

[55] R. Munroe. Lisp. web comic. Available from: http://xkcd.com/224/.

[56] R. Munroe. Lisp cycles. web comic. Available from: http://xkcd.com/297/.

[57] Unknown. Lost ina seaof parentheses. *c2.com wiki*, 2008. Available from: http://c2.com/cgi/wiki?LostInaSeaofParentheses.

[58] L. Wall. Wherefore art, thou? *The Perl Journal*, 1, 1996. Available from: http://www.linuxjournal.com/article/2070.

[59] W. Watt, David A. & Findlay. *Programming language design concepts*. John Wiley & Sons, Ltd., 2004. Available from: http://books.google.com/books?id=vogP3P2L4tgC.