Clojure

A small introduction

Alex Ott

http://alexott.net

Münster JUG, July 2010

About me...

- Professional software developer & architect
- Working with different programming languages – C/C++, Lisps, Haskell, Erlang, ...
- About 20 years of Lisp experience, including commercial projects
- Author of articles on programming (including Clojure), Emacs, etc.
- Participate in many open source projects, including Incanter, labrepl, etc.
- Maintainer of "Planet Clojure"

Outline

1. What is Clojure?
2. Language basics
3. Interoperability with Java
4. Concurrent programming
5. Clojure in real life
6. Sources of information
7. Examples

What is Clojure?

- Lisp-like language, created by Rich Hickey and announced in 2007th
- Designed to work on existing platforms
- Functional programming language, with immutable data
- Special features for concurrent programming
- Open sourced with liberal license
- Already used in many projects, including commercial

Why new language?

- Lisp, but without compatibility with previous versions
- Immutable data and more support for pure functional programming comparing with other Lisps
- Support for concurrent programing inside language
- Better integration with target platforms: JVM & .Net

Main features

- Dynamically typed language
- Very simple syntax, like in other Lisps (code as data)
- Support for interactive development
- Designed in terms of abstractions
- Metaprogramming
- Multimethods and protocols (in version 1.2)
- Compiled into byte code of target platforms
- Access to big number of available libraries

Differences from other Lisps

- Changes in syntax, less parentheses
- Changes in naming
- More first-class data structures – maps, sets, vectors
- Immutable, by default, data
- Ability to link meta-information with variables and functions
- “Lazy” collections
- There is no reader macros
- Case-sensitive names
- Lack of tail call optimization (JVM’s limitation) – explicit loops `loop/recur`
- Exceptions instead of signals and restarts

Base data types

- Integer numbers of any size – 14235344564564564564
- Rational numbers – 26/7
- Real numbers – 1.2345 and BigDecimals – 1.2345M
- Strings (String class from Java) – "hello world"
- Characters (Character from Java) – \a, \b, ...
- Regular expressions – #"[abc]*"
- Boolean – true и false
- nil – the same as null in Java, not an empty list as in Lisp
- Symbols – test, var1, ...
- Keywords – :test, :hello, ...

Data structures

- Separate syntax for different collections:
  - Lists – `(1 2 3 "abc")`
  - Vectors – `[1 2 3 "abc"]`
  - Maps – {:key1 1234 :key2 "value"}
  - Sets – #{:val1 "text" 1 2 10}
- Sequence – abstraction to work with all collections (including classes from Java/.Net)
- Common set of functions to work with sequences
- Lazy operations on sequences
- Vectors, maps, and sets are functions – to simplify access to data in them
- Persistent collections
- Transient collections – performance optimization
- `defstruct` – optimization of map to work with complex data structures

Syntax

- Very simple, homoiconic syntax – program is usual data structure
- Special procedure (reader) process text and produce data structures
- All objects represent themselves except symbols and lists
- Symbols link value with “variable”
- Lists represent forms, that could be:
  - Special form – `def`, `let`, `if`, `loop`, ...
  - Macros
  - Function call or expression, that could be used as function (maps, keywords, returned functional values, ...)
- To get list as-is you need to “quote” it (with `quote` or ’)

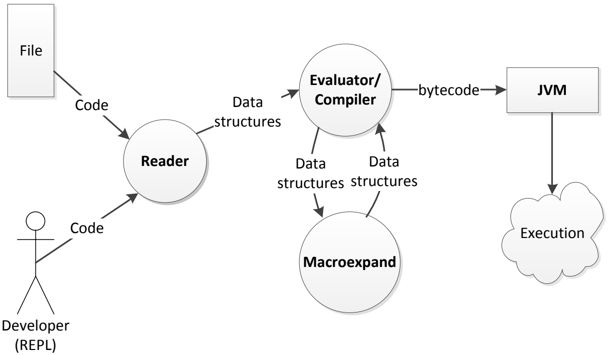
Code example

```
(defn fibo
  ([] (concat [1 1] (fibo 1 1)))
  ([a b]
     (let [n (+ a b)]
       (lazy-seq (cons n (fibo b n))))))
(take 100000000 (fibo))

(defn vrange2 [n]
  (loop [i 0
         v (transient [])]
    (if (< i n)
      (recur (inc i) (conj! v i))
      (persistent! v))))
```

Life cycle of Clojure program

- Code is read from file, or REPL, or from other code
- Reader transforms program into data structures
- Macros are expanded
- Resulting code is evaluated/compiled
- Produced bytecode is executed by JVM



Functions

- Functions are first-class objects
- Function can have variable number of arguments
- Function's definition: defn – top-level functions, fn – anonymous functions
- Simplified syntax for anonymous functions – #(code). For example:
  (map #(.toUpperCase %) ["test" "hello"])
  (map #(vector %1 %2) [1 2 3] [4 5 6])
- You can specify tests & pre/post-conditions. For example,

```
(defn constrained-sqr [x]
  {:pre  [(pos? x)]  :post [(> % 16), (< % 225)]}
  (* x x))
```

- Each function is compiled into separate Java class
- Each function implements Runnable interface

Metaprogramming & macros

- Macros receives source code and returns new code, that will compiled
- Big part of language is written using macros
- Variable number of arguments (same as functions)
- Code could be generated with list functions, but it's much handy to use quasi-quote – ‘ and substitution operators – , and `~@`
- The # suffix in names is used to generate unique names
- `macroexpand-1` & `macroexpand` are used to debug macros

Macros examples (1)

The standard macros `when`:

```
(defmacro when
  [test & body]
  (list 'if test (cons 'do body)))
```

when used as:

```
(when (pos? a)
   (println "positive") (/ b a))
```

will expanded into:

```
(if (pos? a)
  (do
    (println "positive")
    (/ b a)))
```

Macros examples (2)

The standard macros `and`

```
(defmacro and
  ([] true)
  ([x] x)
  ([x & next]
   `(let [and# ~x]
      (if and# (and ~@next) and#))))
```

will expanded into different code, depending on number of arguments:

```
user> (macroexpand '(and ))          ==> true
user> (macroexpand '(and (= 1 2))) ==> (= 1 2)
user> (macroexpand '(and (= 1 2) (= 3 3))) ==>
(let* [and__4457__auto__ (= 1 2)]
  (if and__4457__auto__
     (clojure.core/and (= 3 3))
     and__4457__auto__))
```

Polymorphism & multimethods

- Extensibility
- Multimethods aren't bound exclusively to types/classes
- Dispatching is performed using results of user-specified dispatch function
- Dispatching on several parameters
- Differs from CLOS – absence of `:before`, `:after`, ...
- `defmulti` – is analog of `defgeneric` in Common Lisp
- Defined as `(defmulti func-name dispatch-fn)` + set of implementations (via `defmethod`)

Multimethods examples (1)

Simple dispatching using data type/class:

```
(defmulti m-example class)
(defmethod m-example String [this]
  (println "This is string '" this "'"))
(defmethod m-example java.util.Collection [this]
  (print "This is collection!"))
```

and we'll get:

```
(m-example "Hello") ==> "This is string 'Hello'"
(m-example [1 2 3]) ==> "This is collection!"
(m-example '(1 2 3)) ==> "This is collection!"
```

Multimethods examples (2)

```
(defmulti encounter
     (fn [x y] [(:Species x) (:Species y)]))
(defmethod encounter [:Bunny :Lion] [b l] :run-away)
(defmethod encounter [:Lion :Bunny] [l b] :eat)
(defmethod encounter [:Lion :Lion] [l1 l2] :fight)
(defmethod encounter [:Bunny :Bunny] [b1 b2] :mate)
(def b1 {:Species :Bunny})
(def b2 {:Species :Bunny})
(def l1 {:Species :Lion})
(def l2 {:Species :Lion})

(encounter b1 b2) ==> :mate
(encounter b1 l1) ==> :run-away
(encounter l1 b1) ==> :eat
(encounter l1 l2) ==> :fight
```

Protocols & Datatypes

- Introduced in version 1.2 (will released shortly)
- Much faster than multimethods
- Allows to write “Clojure in Clojure”
- Dispatching is done only on data types
- Similar to type classes in Haskell
- Java interface is created for each protocol
- `defrecord` & `deftype` define new data types
- `extend-protocol` & `extend-type` bind protocol(s) with data types (not only with defined in Clojure!)
- `reify` is used to implement protocols and interfaces for “once-used types”

Protocol's examples

```
(defprotocol Hello "Test of protocol"
  (hello [this] "hello function"))

(defrecord B [name] Hello
  (hello [this] (str "Hello " (:name this) "!")))
(hello (B. "User")) ==> "Hello User!"

(extend-protocol Hello String
              (hello [this] (str "Hello " this "!")))
(hello "world") ==> "Hello world!"

(extend-protocol Hello java.lang.Object
              (hello [this] (str "Hello '" this
                     "'! (" (type this) ")")))
(hello 1) ==> "Hello '1'! (class java.lang.Integer)"
```

Miscellaneous

- Metadata – #^{} in 1.0 & 1.1, or simply ^{} in 1.2
  - Optional type specification (type hints) – #^Type or ^Type
  - You can specify tests directly in function's declaration
  - Scope management
  - Access and change of metadata from code
  - Don't change value equality
- Namespaces:
  - are first-class objects
  - are used to organize code into libraries
- Data destructuring (function arguments, etc.)

```
(let [[a b & c :as e] [1 2 3 4]] [a b c e])
   ==> [1 2 (3 4) [1 2 3 4]]
(let [{:keys [a b c] :as m :or {b 3}} {:a 5 :c 6}]
  [a b c m]) ==> [5 3 6 {:a 5 :c 6}]
```

Interoperability with Java

- Two-way interoperability with target platform:
  - Creation of instances – `new` or `Class.`
  - Call of Java methods – `.`, `..`, `doto`
  - Class & interface generation – `gen-class`, `gen-interface`, or `proxy` (anonymous class)
- You can execute Clojure code in Java programs
- Separate functions to work with Java arrays:
  - `make-array` – array creation
  - `aget`/`aset` – access to array's elements
  - `amap`/`areduce` – iteration over array's elements
- `memfn` allows to use Java methods as arguments of `map`, `filter`, etc.
- The `set!` special form to set values inside class
- Generation and catch of exception – `throw` & `catch`

Interoperability examples

- Instance creation:

```
(new java.util.Date) <==> (java.util.Date.)
```

- Call of Java methods and access to data:

```
(.substring "Hello World" 0 5) ==> "Hello"
(. "Hello World" substring 0 5) ==> "Hello"
Math/PI  ==> 3.141592653589793
(Integer/parseInt "42") ==> 42
```

- .. is used to chained calls:

```
(.. System getProperties (get "os.name")) ==> "Mac
System.getProperties().get("os.name")
```

- doto – call of several methods for one object:

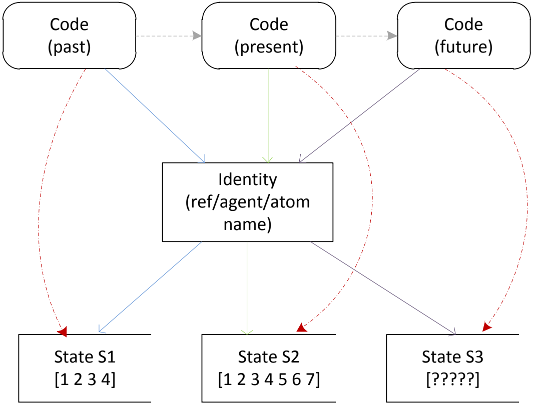
```
(doto (java.util.HashMap.)
 (.put "a" 1) (.put "b" 2))
```

Concurrent programming

- Special language features, that provides data mutation:
    - Refs – synchronous, coordinated change
    - Agents – asynchronous, not coordinated
    - Atoms – synchronous, not coordinated
    - Vars – local for current thread
    - `@` (`deref`) is used to get access to data
- Parallel code execution
    - `future`
    - `pmap`, `pvalues` & `pcalls`
    - Native threads
- Synchronization with `promise`

State and Identity

- Clojure separates concepts of state and identity
- State (value) isn't changed!
- More information at http://clojure.org/state



Refs & STM

- Coordinated change of data from several threads
- Software Transaction Memory provides atomicity, consistency, and isolation
- Changes could be made only inside transaction
- You can add function for data validation
- Ability to add watcher function

```
(def counters (ref {}))
(defn get-counter [key]
  (@counters key 0))
(defn increment-counter [key]
  (dosync
   (alter counters assoc key
          (inc (@counters key 0)))))
```

Agents

- Asynchronous change of data – "fire and forget"
- Thread pools for data update functions: `send` – for "fast" functions, and `send-off` – for "heavyweight" functions (separate thread pool)
- `send` & `send-off` get function that will applied to agent's current state
- Validators and watchers
- You can wait until finish of all updates

```
(def acounters (agent {}))
(defn increment-acounter [key]
  (send acounters assoc key
        (inc (@counters key 0))))
(defn get-acounter [key]
  (@acounters key 0))
```

Vars & Atoms

- Atoms
  - Synchronous data change without coordination
  - Data are changed by applying function to current state of atom
  - Function could be called several times
  - Function shouldn't have side effects!
- Vars
  - Provide data change only in current thread
  - `binding` links new values with symbols
  - Change affects all functions, called from current thread
  - Be careful with lazy sequences!

```
(def *var* 5)
(defn foo [] *var*)
(foo) ==> 5
(binding [*var* 10] (foo)) ==> 10
```

Parallel code execution

- `future`
  - executes given code in separate thread
  - `@` is used to get results of code execution
  - `@` blocks current thread, if results aren't available yet
  - results of code execution are cached
- `promise`
  - is used for synchronization between parts of programs
  - `deliver` sets value of `promise`
  - `@` reads value, or blocks execution if value isn't set yet
- `pmap`, `pvalues` & `pcalls` are used for "heavyweight" operations, that could be performed in parallel

Clojure in real life

- Existing infrastructure:
  - IDEs
  - Build tools
  - Debugging and related tools
  - Libraries
  - Code repositories
- Clojure/core – commercial support & consulting
- Clojure is used in commercial projects since 2008
- Sources of information

Infrastructure: IDEs and build tools

- Supported by most of popular IDEs:
  - Emacs + SLIME (most popular now)
  - VimClojure
  - NetBeans
  - Eclipse
  - IntelliJ IDEA
- Build tools:
  - Clojure support in Maven and Ant
  - Leiningen – written in Clojure, extensible and very simple

Infrastructure: libraries & repositories

- Simple access to big number of existing Java libraries
- Clojure-specific libraries:
    - Clojure-Contrib – “semi-standard” libraries
    - Compojure, Ring, Scriptjure – Web development
    - ClojureQL – databases
    - Incanter – R-like environment and libraries
    - other – see at `http://clojars.org`
- Repositories:
    - build.clojure.org
    - clojars.org

Commercial projects

- FlightCaster – machine learning, etc.
- Relevance, Inc. – consulting
- Runa – marketing services
- Sonian Networks – archiving solutions
- BackType – social media analytics
- DRW Trading Group – trading and finance
- DocuHarvest project by Snowtide Informatics Systems, Inc.
- ThorTech Solutions – scalable, mission-critical solutions
- TheDeadline project by freiheit.com (http://www.freiheit.com/)
- and many more. . .

Sources of information (1)

- Main sites:
  - Site of the language – http://clojure.org
  - Planet Clojure – http://planet.clojure.in
  - the #clojure IRC channel at freenode.net
  - The labrepl project (http://github.com/relevance/labrepl) – learning environment
  - Try-Clojure (http://www.try-clojure.org/) – you can execute Clojure code via Web-browser
  - http://clojuredocs.org/ – documentation and examples
- German resources:
  - Book in German will released in 2010 fall (http://www.clojure-buch.de/)
  - clojure-de mailing list (http://groups.google.com/group/clojure-de)
  - videos about Clojure in German (http://www.rheinjug.de/videos/gse.lectures.app/Talk.html#Clojure)

Sources of information (2)

- Books:
    - Programming Clojure – 2009th, Clojure v. 1.0
    - Practical Clojure. The Definitive Guide – 2010th, Clojure, v. 1.2
    - The Joy of Clojure (beta)
    - Clojure in Action (beta)
    - Clojure Programming on Wikibooks
    - Clojure Notes on RubyLearning
- Video-lectures & screencasts
- Clojure user groups around the world
- Clojure study courses (on-site in USA & Europe, or online)

Examples. Hadoop

- Hadoop word count example is only lines of code (using clojure-hadoop), instead dozens for Java...
- All low-level details are handled by macros

```
(ns clojure-hadoop.examples.wordcount3
  (:import (java.util StringTokenizer)))

(defn my-map [key value]
  (map (fn [token] [token 1])
       (enumeration-seq (StringTokenizer. value))))

(defn my-reduce [key values-fn]
  [[key (reduce + (values-fn))]])
```

Examples. Cascalog

- Domain specific language to query data in Hadoop with joins, aggregates, custom operations, subqueries, etc.
- Interactive work with data, using Clojure
- Available from http://github.com/nathanmarz/cascalog
- Example: query for persons younger than 30:

```
(?<- (stdout) [?person ?age] (age ?person ?age)
               (< ?age 30))
```

Examples. Concurrency

Ants:

- 320 lines of code, with commentaries and documentation
- Massive parallel execution without explicit threading
- Many independent objects
- Graphical interface

Cluster analysis with K-means algorithm:

- Each cluster represented by agent
- Automatically uses all available cores
- `http://www.informatik.uni-ulm.de/ni/staff/HKestler/Reisensburg2009/talks/kraus.pdf`

Thank you

Questions