Erlang vs. Clojure

Apples vs. oranges - they both taste good

Basic stuff

```erlang
%% functions
F = fun(X, Y) -> X + Y.
1> F(1, 10).
11

if %% conditionals
     10 > 5 -> foo
     true -> bar
end.

%% local scope??
2> [_, _, Tail] = lists:seq(0,9).
[0,1,2,3,4,5,6,7,8,9]
3> Tail.
[2,3,4,5,6,7,8,9]

%% lamda function:
4> fun(X) -> 10 + X end (42).
52

%% not really that funny:
5> lists:reverse("hello world").
"dlrow olleh"
```

```clojure
;; Functions

(defn f [x y]
  (+ x y))
;; #'user/f

(f 1 10)
;; 11

;; Conditionals

(if (> 10 5) :foo :bar)
;; :foo

;; Local scope

(let [a (range 10)]
  (rest (rest a)))
;; (2 3 4 5 6 7 8 9)

;; Lambda functions

(#(+ 10 %) 42)
;; 52

;; Funny things

(->> "Hello world" reverse (apply str))
;; "dlrow olleH"
```

Sorting

Total ordering of types!

```
1> lists:sort([1, [1,2,4], atomar, -2,
{hej}, [1, 2, 2], 6, {heap}, {hea}, a]).

[-2,1,6,a,atomar,{hea},{heap},{hej},
[1,2,2],[1,2,4]]
```

```
;; Can do sorting using Javas comparable

(sort [1 5 2])
;; (1 2 5)

;; Not everything is comparable in Java land

(> 10 :foo)
;; clojure.lang.Keyword cannot be cast to java.
lang.Number
;;  [Thrown class java.lang.ClassCastException]

;; But we can filter it out

(defn equals-class? [clazz]
  (fn [obj]
    (= clazz (class obj))))

(filter (equals-class? java.lang.String) [10
"foo" 42 "bar"])
;; ("foo" "bar")
(filter (equals-class? java.lang.Long) [10
"foo" 42 "bar"])
;; (10 42)
```

Pattern matching

```
numbers(1) -> one;
numbers(2) -> two;
numbers(X) when X > 2 ->
    many.


numbers_case(X) ->
    case X of
      1 -> one;
      2 -> two;
      _ -> many
    end.
```

```
(use ['clojure.core.match :only ['match]])

(defn numbers [x]
  (match [x]
         [1] :one
         [2] :two
         [_] :many))
(numbers 1)
;; :one

;; Define defm macro
(defmacro defm [name args & body]
  `(defn ~name ~args (match ~args ~@body)))

(defm numbers-2 [x]
  [1] :one
  [2] :two
  [_] :many)
(numbers-2 2)
;; :two

;; case is more idiomatic
(defn numbers-3 [x]
  (case x
        1 :one
        2 :two
        :many))
(numbers-3 42)
;; :many

;; Can do much more advanced matches
```

Destructering

```
deconstruction([_,[_,{_,What}]]) ->
      What.

1> deconstruction([{1, banan}, [liste, {1,
tupel}]]).
tupel
```

```
(let [list [[1 :banan] [:liste [:1 :
tupel]]]
      [_ [_ [_ what]]] list]
  what)
;; :tupel

(defn foo [[_ y _]] (inc y))
(foo [10 20 30])
;; 21

(let [m {:foo [10 [1 2]] :bar "Hello"}
      {[_ deep-value] :foo} m]
  deep-value)
;; [1 2]
```

Tail call optimization

Part of the language. An example:

```
map(F, L) ->
      map(F, [], L).

map(F, Acc, []) ->
      lists:reverse(Acc);
map(F, Acc, [H|Tail]) ->
      map(F, [F(H) | Acc ], Tail).
```

Just put the recursive call last!

```
;; Explicitly ask for TCO

(loop [i 1]
  (if (> i 10)
    :done
    (if (> i 5)
      (recur (inc i))
      (recur (+ i 2)))))
;; :done

(loop [i 1]
  (if (> i 10)
    :done
    (if (> i 5)
      (do (recur (inc i)) (println "Hej"))
      (recur (+ i 2)))))

;; Can only recur from tail position
;;  [Thrown class java.lang.
UnsupportedOperationException]

(defn loop-fn [i]
  (if (> i 10)
    :done
    (recur (inc i))))
(loop-fn 1)
;; :done
```

List comprehensions

```
%% list comprehensions are
%% a clean, compact way to process
%% lists.

1> [ X * 2 || X <- lists:seq(1,10) ].
[2,4,6,8,10,12,14,16,18,20]

2> [ {X, Y} || X <- lists:seq(1,3), Y <-
lists:seq(1,3) ].
[{1,1},{1,2},{1,3},{2,1},{2,2},{2,3},
{3,1},{3,2},{3,3}]
```

```
;; List comprehensions are just simple for
loops (one less concept)

(for [i (range 100) :when (< i 10)]
 (* 2 i))
;; (0 2 4 6 8 10 12 14 16 18)

;; Products

(for [i (range 100) j (range 3)
      :when (< i 10)] [j (* 2 i)])
([0 0] [1 0] [2 0] [0 2] [1 2] [2 2] [0 4]
[1 4] [2 4] [0 6] [1 6] [2 6] [0 8] [1 8]
[2 8] [0 10] [1 10] [2 10] [0 12] [1 12]
[2 12] [0 14] [1 14] [2 14] [0 16] [1 16]
[2 16] [0 18] [1 18] [2 18])
```

Type system

Ouch!

The compiler does NO type checks.

There is a language extension
that helps:
* Static analysis /w Dialyzer
* For documentation (EDoc)

Example:

```
-spec foo({X, integer()}) -> X when X :: atom()
      ; ([Y]) -> Y when Y :: number().
```

```
;; Type system inherited from Java

(class 1)
;; java.lang.Long

(class "Hello world")
;; java.lang.String

;; Strong emphesis on few interfaces
;; with many functions

(rest [1 2 3])
;; (2 3)

(map inc #{1 2 3})
;; (2 3 4)

(map inc '(1 2 3))
;; (2 3 4)
```

Data structures

dict, orddict, proplists - all list based. O(N)/O(1) insert, O(N) search, lookup.

gb_trees - (AVL trees with normal O(log n) behaviour.

ETS, DETS - lookup: O(1), (log N) for resp. unsorted, sorted. As sets, bags, etc.

Process dictionary: fast, destructive. No message copying. Beware of this temptation!!!

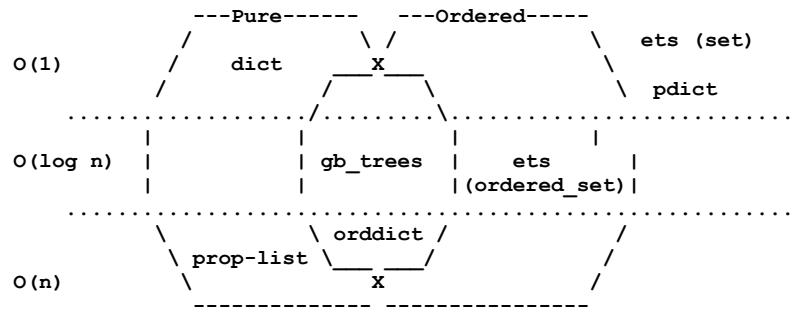


Diagram src: https://github.com/eriksoe/AGttES

```
;; Sets
(#{:foo :bar :baz} :not-here)

;; Maps
({:foo 1, :bar 42} :foo)

;; Vectors
[1 2 42]

;; Lists
'(3 2 1)

;; structures are faster if arguments are known

(defstruct test-struct :member1 :member2 :member3)

(assoc (struct test-struct :foo :bar :baz) :test :
fest)

;; Records are classes

(defrecord TestRecord [member1 member2 member3])

(TestRecord. :foo :bar :baz)

;; It is more idiomatic to just store data in maps

{:uri "http://..."
 :input "dsfgsgsfd"
 :type :POST}
```

Producer/consumer

```
start_consumer() ->
    spawn(fun() -> consumer_loop(queue:new()) end).

consumer_loop(Queue) ->
    receive
      {add, Product} -> consumer_loop(queue:in
(Product, Queue));
      {get, From} ->
          case queue:out(Queue) of
            {{value, Product}, NewQueue} ->
                From ! {ok, Product},
                consumer_loop(NewQueue);
            {empty, NewQueue} -> From ! {error,
empty},
                                 consumer_loop(NewQueue)
          end;
      _Other -> consumer_loop(Queue)
    end.
```

```
(def food-agent (agent nil))
;;#'user/food-agent

(defn consumer [key ref old new]
  (println
   (case new
       :meat "Yummy that tasted good"
       "I do not want to consume that")))
;; #'user/consumer

(add-watch food-agent
 "a key representing the watch"
 consumer)
;; #<Agent@1b7adb4a: nil>

(defn change-food-stuff [current-food new-food]
  new-food)
;; #'user/change-food-stuff

(send food-agent change-food-stuff :meat)
;; Yummy that tasted good
;; #<Agent@1b7adb4a: :meat>

(send food-agent change-food-stuff :banana)
;; I do not want to consume that
;;#<Agent@1b7adb4a: :banana>
```

Lazy eval / infinite sequences

Erlang does not have infinite sequences built in :(

```
1> Range = lazy_eval:range(fun(X) ->
      lazy_eval:inc(lazy_eval:even(X))
           end, 0).
#Fun<lazy_eval.0.6128643>

2> lazy_eval:take(5, Range).
{#Fun<lazy_eval.0.6128643>,
      [1,3,5,7,9]}

3> {R2, _} = lazy_eval:take(5, Range).
{#Fun<lazy_eval.0.6128643>,
      [1,3,5,7,9]}

4> lazy_eval:take(5, R2).
{#Fun<lazy_eval.0.6128643>,
      [11,13,15,17,19]}
```

```
;; Core component of language
;; Raises level of abstraction

(def evens (filter even? (range)))
(take 5 evens)
;; (0 2 4 6 8)

(map inc '(1 2 3))
;; (2 3 4)

(take 5 (map inc evens))
;; (1 3 5 7 9)
```

Lazy eval / infinite sequences

but - slightly more elaborate, and not as powerful:

```
even(X) ->
    2*X.

inc(X) ->
    X + 1.

range(F, Num) ->
    fun() -> [F(Num)|range(F, Num+1)] end.

take(NumItems, Range) ->
    generate(Range, NumItems, []).

generate(_Fun, 0, Acc) ->
     lists:reverse(Acc);
generate(Fun, ToGo, Acc)  ->
    [Val|Func] = Fun(),
    generate(Func, ToGo - 1,
           [Val | Acc]).
```

```
;; Core component of language
;; Raises level of abstraction

(def evens (filter even? (range)))
(take 5 evens)
;; (0 2 4 6 8)

(map inc '(1 2 3))
;; (2 3 4)

(take 5 (map inc evens))
;; (1 3 5 7 9)
```

Datagram parsing

```
<<4:4,
  HLen:4,
  SrvcType:8,
  TotLen:16,
  ID:16,
  Flgs:3,
  FragOff:13,
  TTL:8, Proto:8,
  HdrChkSum:16,
  SrcIP:32,
  DestIP:32,
  RestDgram/binary>> = list_to_binary(
hex_string:hexstr_to_list(
"4500001f000040004011f3cc0a000067dd805f1a6
296b636000ba981")).

5> SrcIP.
167772263
11> <<A:8, B:8, C:8, D:8>> = <SrcIP:32>>.
<<10,0,0,103>>
```

```
(defn strip-bits [{bit-seq :bit-seq :as
result} name length]
  (assoc result
    name (take length bit-seq)
    :bit-seq (drop length bit-seq)))

(let [bit-seq (->> datagram-packet hex-
to-int-seq int-seq-to-bit-seq)]
  (-> {:bit-seq bit-seq}
      (strip-bits :ip-version 4)
      (strip-bits :hlen 4)
      ;; ...
      ))

{:hlen (0 0 0 1), :ip-version (0 0 0
1), :bit-seq (0 0 1 1  ... )}
```

Many processes

```
1> timer:tc(fun many_processes:max/1,
[1000]).
Process spawn time = 0.0 (4.0)
microseconds
{5062,ok}

2> timer:tc(fun many_processes:max/1,
[10000]).
Process spawn time = 4.0 (5.1)
microseconds
{50741,ok}

3> timer:tc(fun many_processes:max/1,
[100000]).
Process spawn time = 7.7 (6.7)
microseconds
{663350,ok}

4> timer:tc(fun many_processes:max/1,
[1000000]).
Process spawn time = 7.62 (7.031)
microseconds
{6960810,ok}
```

```
;; Threads are native Java threads ;-(

;; Lightweight threads in Java

;; agents, atoms, etc. all extremely
effecient

(def iterations 100000)

(def counter (agent iterations))

(def collector (atom []))

(defn handler [key ref old new]
  (swap! collector conj new))

(add-watch counter :handler handler)

(time
 (doseq [i (range iterations)]
   (send counter dec)))

;; "Elapsed time: 533.636 msecs"
;; nil
```

Encapsulation

Code:
Modules(containers and namespaces) and functions.

Data:
processes and messsage parsing.

Immutable data structures.

Pure functions.

Encapsulation via agents, refs, atoms (all mutable state).

Immutable data structures

Low coupling / complexity

Pure functions

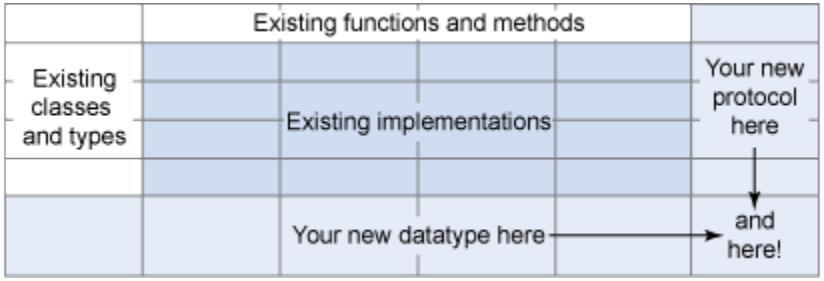
Servlets / whatever the Java world comes up with

Expression problem

The **Expression Problem** is a term used in discussing strengths and weaknesses of various programming paradigms and programming languages. The expression problem can be treated as a use case in programming language design.

Philip Wadler coined the term:

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).[6]



Conclusion

?