複素環数論レポート

情報科学類二年 江畑 拓哉 (201611350)

1 今回作成したもの

基本的な方針としては Clojure/ClojureScript を用いたプログラミングを行うことを前提として 3 つのプログラムを作成した。編集環境は Manjaro Linux 17.0 ”Gellivara” を用いている。必要と思われるパッケージを以下に列挙しておく。

- jre8-openjdk
- jdk8-openjdk
- leiningen
- git

いずれも、ArchLinux 系列の Linux を用いているならば、 yaourt -S コマンドでインストールが可能である。

2 ClojureScript

clojurescript は javascript のメタ言語となっている。そのため、ブラウザから実行を確認することができる。

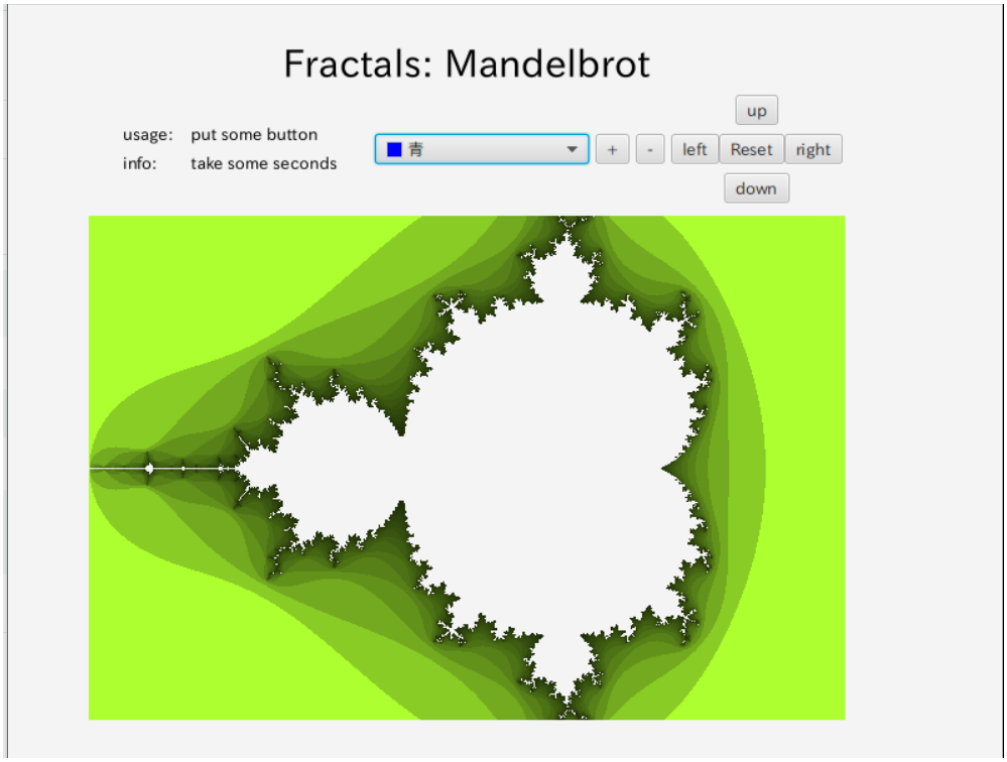
`https://github.com/MokkeMeguru/mandelbrot-cljs-processig`

ここから clone をして index.html をみることで実行を確認することがわかる。主に編集したコードは他の制作物も含めて末尾にまとめて記載するが、mandelbrot-cljs-processig/src/mandelbrot_cljs_app に含まれている。

残念ながら README に書かれているように速度がでないことが問題になっている。更に拡大などの機能も追加しなかった。

3 Clojure

Clojure と Javafx を用いたアプリケーションのサンプルも兼ねてこちらを制作した。速度との両立も兼ね精度はそこまで出すことができなかったが、グラフィカルな動作を行うことができた。

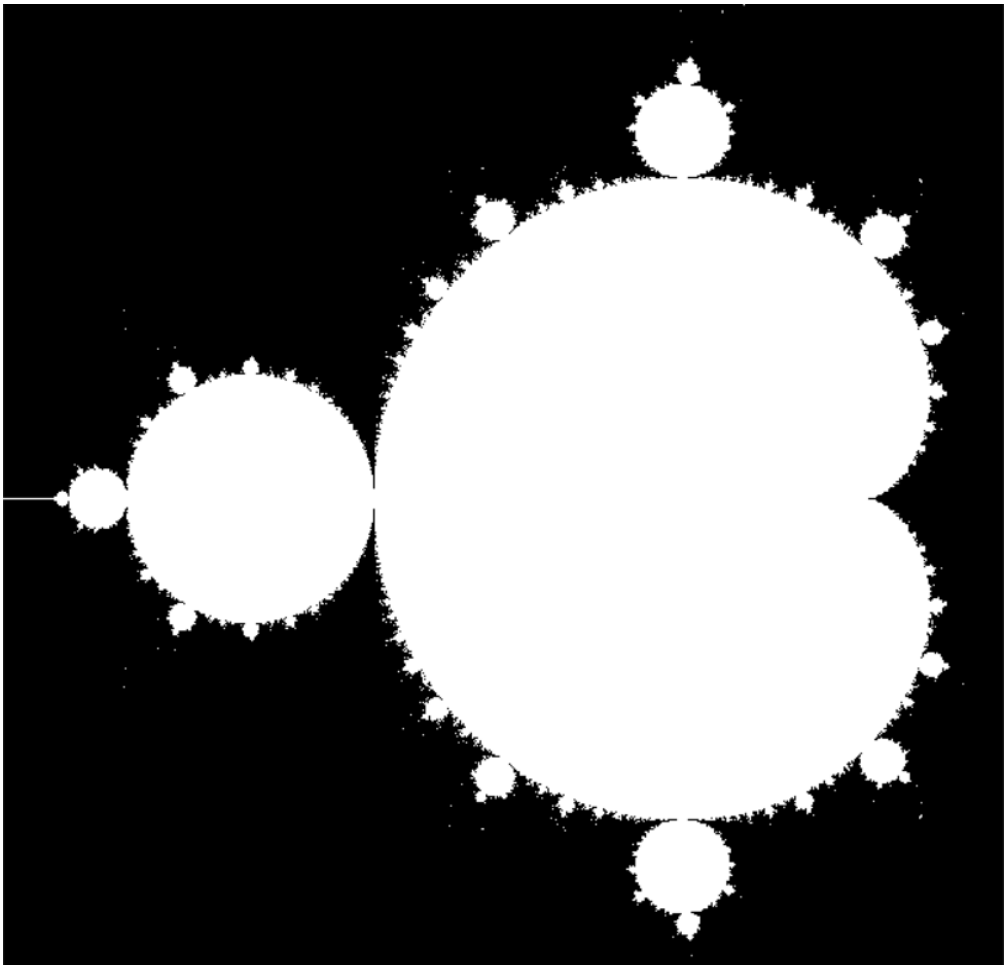


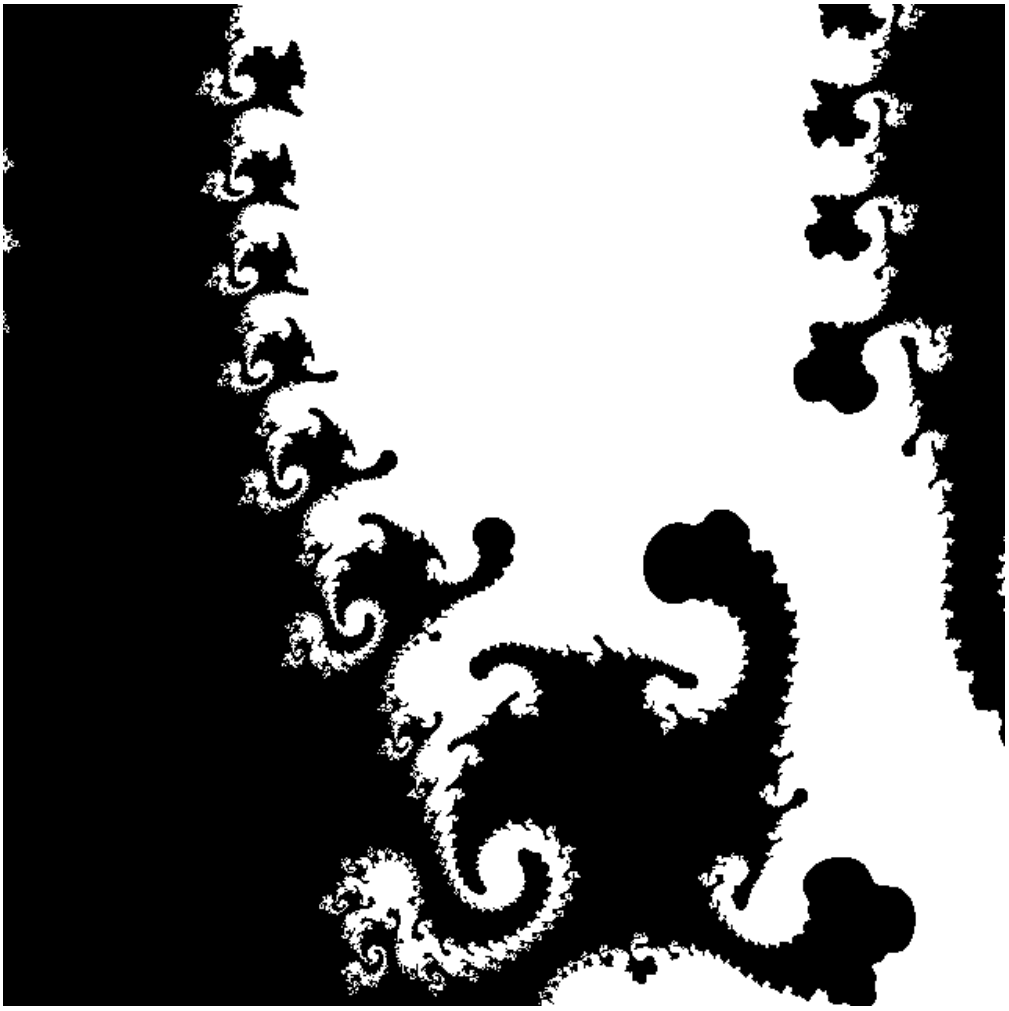
4 clojure × Chapel

速度が足りなかったため、自分の力量不足を感じたため単純に速度の出る Chapel を用いて計算を行った。これにより圧倒的に計算量を増やすことができた。(今回場合実行速度を低下させている要因は、計算結果を描画する段階にあった)

結果としてこれが最も速度が出て安定した結果を出力することができた。しかし Java 側の問題か、Windows 上では実行することができなくなってしまった。

`https://github.com/MokkeMeguru/chapel-clojure-apps`





5 ソースコードなど

以下にそれぞれのソースコードを添付する。

5.1 clojurescript

project.clj

```
(defproject mandelbrot-cljs-app "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
```

```
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [quil "2.6.0"]
                 [org.clojure/clojurescript "1.9.473"]]

  :plugins [[lein-cljsbuild "1.1.5"]]
  :hooks [leiningen.cljsbuild]

  :cljsbuild
  {:builds [{:source-paths ["src"]
             :compiler
             {:output-to "js/main.js"
              :output-dir "out"
              :main "mandelbrot_cljs_app.core"
              :optimizations :none
              :pretty-print true}}]})
```

core.cljs

```
(ns mandelbrot-cljs-app.core
  (:require [quil.core :as q :include-macros true]
            [quil.middleware :as m]))

(def depth 20)

(defn calc [c z]
  (let [cx (:x c)
        cy (:y c)
        x (:x z)
        y (:y z)]
    {:x (+ cx (* x x) (- 0 (* y y)))
     :y (+ cy (* 2 x y))}))

(defn comp-size [c]
  (let [x (:x c)
        y (:y c)]
    (+ (* x x) (* y y))))

(defn mandelbrotbean [c z time]
  (cond
    (< 4 (comp-size z)) nil
    (< time 0) true
    :else (mandelbrotbean c (calc c z) (dec time))))

```

```
26
27 (defn mandelbrotbean? [[a i]]
28   (mandelbrotbean {:x a :y i} {:x 0 :y 0} depth))
29
30 (def cell-size 800)
31
32 (def mandelbrots
33   (doall (filter mandelbrotbean? (for [a (range -2 1 (/ 1.0 cell-size))
34                                        i (range -1 1 (/ 1.0 cell-size))]
35                                    [a i]))))
36 (defn setup []
37   (q/background 255 255 255))
38
39 (defn draw []
40   (q/push-matrix)
41   (q/stroke 50 0 100)
42   (q/scale 0.5 0.5)
43   (q/translate (* 2 cell-size) (* 1 cell-size))
44   (doseq [c mandelbrots]
45     (apply q/point (map #(* % cell-size) c)))
46   (q/pop-matrix)
47   (q/no-loop))
48
49 (q/defsketch mandelbrot-cljs-app
50   :size [(* 1.5 cell-size) (* 1 cell-size)]
51   :setup setup
52   :draw draw)
```

5.2 clojure

project.clj

```
1  (defproject mandelbrot-javafx-clj "0.1.0-SNAPSHOT"
2    :description "FIXME: write description"
3    :url "http://example.com/FIXME"
4    :license {:name "Eclipse Public License"
5              :url "http://www.eclipse.org/legal/epl-v10.html"}
6    :dependencies [[org.clojure/clojure "1.8.0"]]
7    :main mandelbrot-javafx-clj.javafx-init
8    :aot [mandelbrot-javafx-clj.javafx-init]
9    :target-path "target/%s"
10   :profiles {:uberjar {:aot :all}})
```

core.clj

```
(ns mandelbrot-javafx-clj.core
  (:import (javafx.scene.paint Color)
           (javafx.application Application)
           (javafx.stage Stage)
           (javafx.awt.Color)
           (javafx.scene Scene)
           (javafx.scene.layout.Pane)
           (javafx.scene.layout GridPane BorderPane HBox VBox)
           (javafx.geometry Pos Insets)
           (javafx.scene.text Text Font FontWeight)
           (javafx.scene.control Button ColorPicker Label)
           (javafx.event EventHandler Event)
           (javafx.embed.swing SwingFXUtils)
           (javafx.scene.image ImageView)
           (javafx.scene.canvas Canvas)
           )
  (:require [clojure.java.io :as io])
  (:gen-class
   :extends javafx.application.Application))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Here is the function of calculating Mandelbrot sets / returns int RGB array ;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(def width 600)
(def height 400)
(def depth 25)

(defn mandelbrotbean
  "mandelbrotbean \n
  x means x-position, y means y-position, \n
  opacity means javafx.scene.paint.Color instance \n
  Return: javafx.scene.paint.Color instance"
  [^clojure.lang.PersistentList pos
   ^javafx.scene.paint.Color color]
  (let [cx (first pos)
        cy (second pos)]
    (loop [x 0 y 0 times 1]
      (cond
```

```
41          (== times depth) (.deriveColor color 0.0 1.0 0.0 0.0)
42          (> (+ (* x x) (* y y)) 4) (.deriveColor color 0.0 1.0 (* 4 (/ 1 times)) 1.0)
43          :else (recur (+ (* x x) (* -1.0 y y) cx)
44                       (+ (* 2.0 x y) cy)
45                       (inc times))))))
46
47  (defn write-image [int_list]
48    (let [img (java.awt.image.BufferedImage. width height (java.awt.image.BufferedImage/TYP
49      (.setRGB img 0 0 width height  (int-array int_list) 0 width)
50      img))
51
52  (defn mandelbrot
53    "get mandelbrot list \n
54    pos means {:x-min :x-max :y-min :y-max}, \n
55    color means javafx.scene.paint.Color inst(def ex (atom nil))
56    Return: Bufferedimage "
57    [^clojure.lang.PersistentArrayMap pos
58     ^javafx.scene.paint.Color color]
59    (let [x-min (:x-min pos)
60          x-max (:x-max pos)
61          y-min (:y-min pos)
62          y-max (:y-max pos)
63          pos-data (for [k (range y-min y-max (/ (- y-max y-min) height))
64                         i (range x-min x-max (/ (- x-max x-min ) width))] [i k])]
65      (write-image (doall (for [pos-list pos-data]
66                            (let [precolor (mandelbrotbean pos-list color)
67                                  recolor (java.awt.Color. (float (.getRed precolor))
68                                                           (float (.getGreen precolor))
69                                                           (float (.getBlue precolor))
70                                                           (float (.getOpacity precolor)))]
71                              (.getRGB recolor)))))))
72
73  (defn write-image-to-bufferedimage [^:java.awt.image.BufferedImage img]
74    (SwingFXUtils/toFXImage img nil))
75
76  (defn mandelbrot-javafx [^clojure.lang.PersistentArrayMap pos
77                           ^javafx.scene.paint.Color color]
78    [pos color]
79    (let [x-min (:x-min pos)
80          x-max (:x-max pos)
81          y-min (:y-min pos)
82          y-max (:y-max pos)
83          pos-data (for [k (range y-min y-max (/ (- y-max y-min) height))
84                         i (range x-min x-max (/ (- x-max x-min ) width))] [i k])]
```

```
85      (write-image-to-bufferedimage
86       (write-image (doall (for [pos-list pos-data]
87                             (let [precolor (mandelbrotbean pos-list color)
88                                   recolor (java.awt.Color. (float (.getRed precolor))
89                                                            (float (.getGreen precolor))
90                                                            (float (.getBlue precolor))
91                                                            (float (.getOpacity precolor))
92                                                            (.getRGB recolor))))))))
93  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
94  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
95  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
96  ;;;; You set below data, which is atomic data ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
97  (def initial-state {:pos {:x-mix -2 :x-max 1 :y-min -1 :y-max 1}
98                      :color (Color/GREENYELLOW)
99                      :root-stage? true
100                     :redraw? false})
101
102 (def initial-data (mandelbrot-javafx {:x-min -2 :x-max 1 :y-min -1 :y-max 1} (Color/GREE
103
104 ;; initialize atomic data
105 (defonce data-data (ref initial-data))
106 (defonce data-state  (ref initial-state))
107 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
108 ;; You should add some button scene h-box and etc below ;;;;;;;;;;;;;;;;;;;;;;;;;;
109 (defn set-text [^java.lang.String strings]
110   (Text. strings))
111
112 (defn data-reset [^clojure.lang.PersistentArrayMap pos
113                   ^javafx.scene.paint.Color color
114                   ^java.lang.Boolean root-stage
115                   ^clojure.lang.Ref new-data]
116   (println "Call Function: data-reset")
117   (dosync
118    (ref-set
119     data-state {:pos pos :color color
120                 :root-stage? root-stage :redraw? false})
121    (ref-set
122     data-data new-data)))
123
124 (defn set-image []
125   (let [redraw (:redraw? @data-state)
126         pos    (:pos @data-state)
127         color (:color @data-state)
128         root-stage (:root-stage? @data-state)]
```

```
129     (if-not redraw
130        @data-data
131       (do
132         (println "Call Function: mandelbrot")
133         (data-reset pos color root-stage (mandelbrot-javafx pos color))
134         (println "End Function: Imageview")
135          @data-data))))
136
137
138 (def image (ImageView. ))
139 (defn set-writable-image []
140   (let []
141     (.setImage image (set-image))
142     image))
143
144 (defn event-button-selected [^clojure.lang.Keyword keyword]
145   (let [data-states @data-state
146         pos (:pos data-states)
147         color (:color data-states)
148         root-stage? (:root-stage? data-state)
149         redraw? (:redraw? data-state)
150         x-min (:x-min pos)
151         x-max (:x-max pos)
152         x-quarter (/ (- x-max x-min) 4)
153         y-min (:y-min pos)
154         y-max (:y-max pos)
155         y-quarter (/ (- y-max y-min) 4)]
156     (println "call: " keyword)
157     (cond
158       (= keyword :Reset)
159       (do
160         (dosync
161          (println "call: " keyword)
162          (ref-set data-state
163                   {:pos {:x-min -2 :x-max 1 :y-min -1 :y-max 1} :color (Color/GREENYELLO
164                    :root-stage? false :redraw? true})
165          (set-writable-image)))
166       (= keyword :right)
167       (do
168         (println "call: " keyword)
169         (dosync
170          (ref-set data-state
171                   {:pos {:x-min (+ x-min x-quarter)
172                          :x-max (+ x-max x-quarter)
```

```
                        :y-min y-min :y-max y-max}
                 :color color
                 :root-stage? false :redraw? true})
    (set-writable-image)))
(= keyword :left)
(do
  (println "call: " keyword)
  (dosync
   (ref-set data-state
            {:pos {:x-min (- x-min x-quarter)
                   :x-max (- x-max x-quarter)
                   :y-min y-min :y-max y-max}
             :color color
             :root-stage? false :redraw? true})
    (set-writable-image)))
(= keyword :down)
(do
  (println "call: " keyword)
  (dosync
   (ref-set data-state
            {:pos {:x-min x-min :x-max x-max
                   :y-min (+ y-min y-quarter)
                   :y-max (+ y-max y-quarter)}
             :color color
             :root-stage? false :redraw? true})
    (set-writable-image)))
(= keyword :up)
(do
  (println "call: " keyword)
  (dosync
   (ref-set data-state
            {:pos {:x-min x-min :x-max x-max
                   :y-min (- y-min y-quarter)
                   :y-max (- y-max y-quarter)}
             :color color
             :root-stage? false :redraw? true})
    (set-writable-image)))
(= keyword :-)
(do
  (println "call: " keyword)
  (dosync
   (ref-set data-state
            {:pos {:x-min (- x-min x-quarter) :x-max (+ x-max x-quarter)
                   :y-min (- y-min y-quarter)
```

```
                          :y-max (+ y-max y-quarter)}
                   :color color
                   :root-stage? false :redraw? true})
          (set-writable-image)))
      (= keyword :+)
      (do
        (println "call: " keyword)
        (dosync
         (ref-set data-state
                  {:pos {:x-min (+ x-min x-quarter) :x-max (- x-max x-quarter)
                         :y-min (+ y-min y-quarter)
                         :y-max (- y-max y-quarter)}
                   :color color
                   :root-stage? false :redraw? true})
         (set-writable-image))))))

(defn set-button [^clojure.lang.Keyword keyword]
  (let [button (Button. (clojure.string/join (rest (str keyword))))]
    (doto button
      (.setOnAction (proxy [EventHandler] []
                      (handle [ActionEvent]
                        (event-button-selected keyword)))))
    button))

(defn event-color-pickup [^:ColorPicker this]
  (let [new-color (.getValue this)
        pos (:pos @data-state)
        root-stage? (:root-stage? @data-state)
        old-color (:color @data-state)]
    (when (not= new-color old-color)
      (dosync
       (println "Color Change!")
       (ref-set data-state
                {:pos pos :color new-color
                 :root-stage? root-stage? :redraw? true})
       (set-writable-image)))))

(defn set-color-picker []
  (let [color (ColorPicker. (javafx.scene.paint.Color/BLUE))]
    (doto color
      (.setOnAction (proxy [EventHandler] []
                      (handle [ActionEvent]
                        (event-color-pickup color)))))))
```

```
(defn set-border-pane []
  (let [top (set-button :up)
        bottom (set-button :down)
        right (set-button :right)
        left (set-button :left)
        border-pane (BorderPane. (set-button :Reset)
                                 top
                                 right
                                 bottom
                                 left)]
    (do
      (BorderPane/setAlignment top Pos/TOP_CENTER)
      (BorderPane/setAlignment right Pos/CENTER_RIGHT)
      (BorderPane/setAlignment bottom Pos/BOTTOM_CENTER)
      (BorderPane/setAlignment left Pos/CENTER_LEFT))
    border-pane))

(def gridpane (GridPane.))
(defn set-grid-pane []
  (let []
    (doto gridpane
      (.setAlignment Pos/CENTER)
      (.setHgap 10)
      (.setVgap 10)
      (.setPadding (Insets. 25 25 25 25))
      (.add (set-text "usage: " ) 0 0)
      (.add (set-text "put some button") 1 0)
      (.add (set-text "info: ") 0 1)
      (.add (set-text "take some seconds") 1 1)
      )))

(defn set-h-box []
  (let [hbox (doto (HBox. 5.0)
               (.setAlignment Pos/CENTER))
        _  (.add (.getChildren hbox) (set-grid-pane))
        _  (.add (.getChildren hbox) (set-color-picker))
        _  (.add (.getChildren hbox) (set-button :+))
        _  (.add (.getChildren hbox) (set-button :-))
        _  (.add (.getChildren hbox) (set-border-pane))]
    hbox))

(defn set-v-box []
  (let [vbox (doto (VBox. 10.0)
               (.setAlignment Pos/CENTER)
```

```
                ;;(-> .getChildren .add (set-text "development"))
              )
        _ (.add (.getChildren vbox) (doto (Text. "Fractals: Mandelbrot")
                                      (.setFont (Font/font "Verdana" 30))))
        _ (.add (.getChildren vbox) (set-h-box))
        _ (.add (.getChildren vbox) (set-writable-image))]
    vbox))

(defn set-scene []
  " arguments means children \n
  scene
    |- v-box :as set-v-box
         |- text (title) :as set-text
         |- h-box :as set-h-box
             |- grid-pane (view position) :as set-grid-pane
                    |- text (usage) 0 0 :as set-text-prop
                    |- text () 0 1 :as set-text-prop
                    |- text (info) 1 0 :as set-text-prop
                    |- text () 1 1 :as set-text-prop
             |- color-picker ! redraw :as set-color-picker :with event-color-pickup
             |- button (bigger) ! redraw :as set-button
             |- button (smaller) ! redraw :as set-button
             |- border-pane :as set-border-pane
                    |- top - button (up) ! redraw :as set-button
                    |- bottom - button (down) ! redraw :as set-button
                    |- right - button (right) ! redraw :as set-button
                    |- left - button (left) ! redraw :as set-button
                    |- center - button (reset) ! redraw :as set-button
         |- canvas - writable-image (-> pixel-writer -> set-pixels) :as set-writable-imag
         |- text (some-information) :as set-text
  ref
    color - javafx.scene.Color.
    data - int-rgb-list (<- getRGB <- java.awt.color <-get-red/green/blue/opacity )
    root-stop? - force-exit
    redraw? - need of redrawing
  "
  (let []
    (doto (Scene. (set-v-box) (+ 250 width) (+ 200 height)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; NECESSARY UTILITY ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defn force-exit
  " This is closed function \n"
  [root-stage?]
  (reify javafx.event.EventHandler
    (handle [this event]
      (when (not root-stage?)
        (do (println "Closing application")
            (javafx.application.Platform/exit))))))

(defn swap
  " This is change state of initialize \n"
  [root-stage?]
  (dosync (ref-set data-state {:root-stage? root-stage?
                               :pos {:x-min -2 :x-max 1 :y-min -1 :y-max 1}
                               :color (Color/GREENYELLOW)
                               :redraw? false})))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

javafx_init.clj

```
(ns mandelbrot-javafx-clj.javafx-init
     (:require [mandelbrot-javafx-clj.core :as core])
     (:import (javafx.application Application)
              (javafx.stage Stage))
     (:gen-class
   :extends javafx.application.Application))

(defn -start
  " This is javafx-start funnction \n"
  [this ^Stage stage]
  (let []
    (doto stage
      (.setTitle "Fractals: Mandelbrot")
      (.setOnCloseRequest (core/force-exit {:root-stage? false}))
      (.setScene (core/set-scene))
      ;;(core/root-stage {:root-stage? false})
      .show)))

(defn -main
  " This is javafx-launch function \n"
  [& args]
```

```
  (core/swap false)
  (Application/launch mandelbrot_javafx_clj.javafx_init (into-array String [])))
```

5.3 clojure × chapel

project.clj

```
(defproject chapel-clojure-app "0.1.0"
  :description "Fractal: Mandelbrot"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [org.clojure/core.async "0.3.443"]
                 [seesaw "1.4.5"]]
  :profiles {:uberjar {:main chapel-clojure-app.core, :aot :all}
             :dev {:resource-paths ["resources"]}}
  :main chapel-clojure-app.core)
```

chapel

```
use DynamicIters;

config const n = 1600,
  maxIter = 150,
  limit = 4.0,
  chunkSize = 1,
  size = 2.0,
  xstart = -1.5,
  ystart = -1.0;

param bitSize = 8;
type elementSize = uint(bitSize);

proc main() {
  const xdim = 0..#divceilpos(n, bitSize);
  var imageSpace : [0..#n, xdim] elementSize;

  // domain means first
  forall (y, xelem) in dynamic(imageSpace.domain, chunkSize) {
    var buff: elementSize; // declare : var something: type(size);
```

```
21
22      for off in 0..#bitSize {
23        var Zn1, Zn0: complex; // declare complex
24
25        const x = xelem * bitSize + off; // where is x in your logical memory?
26        const complexVal = (size * x/n + xstart) + (size * y/n + ystart) * 1i;
27
28        for 1..maxIter{
29          if((Zn0.re + Zn0.im) ** 2 - (2 *Zn0.re * Zn0.im) > limit) then
30            break;
31
32          Zn1.re = Zn0.re ** 2 - Zn0.im ** 2 + complexVal.re;
33          Zn1.im = 2.0 * Zn0.re * Zn0.im + complexVal.im;
34
35          Zn0.re = Zn1.re;
36          Zn0.im = Zn1.im;
37        }
38
39        buff <<= 1;
40        if ((Zn0.re + Zn0.im) ** 2 - (2 *Zn0.re * Zn0.im) <= limit) then
41          buff |= 0x1; // draw a black point
42      }
43
44      imageSpace[y, xelem] = buff; // store the pixel
45    }
46
47    var w = openfd(1).writer(iokind.native, locking=false);
48
49  //  w.writef("P4\n");
50  //  w.writef("%i %i\n", n, n);
51    w.write(imageSpace);
52  }
```

core.clj

```
1  (ns chapel-clojure-app.core
2    (:require [clojure.java.io :as io]
3              [clojure.core.async :as async]
4              [seesaw.core :as core]
5              [seesaw.graphics :as g]
6              [seesaw.dev :as dev]
7              [clojure.java.io :refer [output-stream input-stream]]
8              [clojure.pprint :refer (cl-format)])
```

```
  (:import [java.awt.image BufferedImage]
           [javax.swing JFrame JLabel ImageIcon WindowConstants SwingUtilities]
           [java.awt.event KeyEvent]
           [java.awt Graphics2D]
           [java.awt Color])
  (:use [clojure.java.shell :only [sh]])
  (:gen-class))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; init temporary-file ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defn copy-file [^java.io.File file ^java.io.BufferedInputStream stream]
  (with-open [in stream
              out (output-stream file)]
    (io/copy in out))
  file)

(defonce exec-file (if-not (.exists (io/as-file "temp"))
                     (let [file (java.io.File/createTempFile "temp" "")
                           _ (.deleteOnExit file)
                           _ (sh "sh" "-c" (str "chmod +x " file))]
                       (copy-file file
                                  (io/input-stream
                                   (io/resource "my-mandelbrot-chapel"))))
                     (copy-file  (io/file "temp")
                                 (io/input-stream
                                  (io/resource "my-mandelbrot-chapel")))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; draw mandelbrot buffered-image with Chapel ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defonce image-data (ref nil))

(defn get-binary
  " require nothing
    return clojure.lang.PersistentVector "
  []
  @image-data
  )

(defn byte-array->color-array
  " return lazy-seq (int (java.awt.color))"
  [^clojure.lang.PersistentVector byte-array]
  (let [binary-array byte-array
        black (.getRGB Color/BLACK)
        white (.getRGB Color/WHITE)
```

```
53          new-int (pmap (fn [i]
54                          (loop [c 0
55                                 acc (list)]
56                            (if (< c 8)
57                              (if (= 0 (bit-and (bit-shift-right i c) 0x01))
58                                (recur (inc c) (conj acc black))
59                                (recur (inc c) (conj acc white)))
60                              acc)))
61                        binary-array)]
62      (reduce into [] new-int)))
63
64  (defn color-array->buffered-image
65    " return buffered-image"
66    [^clojure.lang.PersistentVector color-array]
67    (let [array (int-array color-array)
68          buffered-image (BufferedImage. 640 640 BufferedImage/TYPE_4BYTE_ABGR)
69          _ (.setRGB buffered-image 0 0 640 640 array 0 640)]
70      buffered-image))
71
72
73  (defn redraw [{:keys [size xstart ystart]}]
74    (let [n " --n=640"
75          size (str " --size=" size)
76          xstart (str " --xstart=" xstart)
77          ystart (str " --ystart=" ystart)
78          program (io/file exec-file)
79          ]
80      (dosync
81       (ref-set image-data  (:out (sh "sh" "-c" (str program xstart ystart size n) :out-enc
82    (println ""))
83
84  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
85  ;; attributes ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
86
87  (defn img []
88    (color-array->buffered-image (byte-array->color-array (get-binary))))
89
90  ;; init state
91  (def initial-pos {:size 2.0 :xstart -1.5 :ystart -1.0})
92  (defonce refpos (ref initial-pos))
93  (defonce _ (redraw {:size 2.0 :xstart -1.5 :ystart -1.0}))
94
95  (add-watch refpos :watcher
96             (fn [key ref old-value new-value]
```

```
                (println "old: " old-value
                         " new: " new-value)))


;; declare move direction
(defn position [n]
  (cond (> n 400) 3
        (> n 200) 2
        :default 1))

(defn set-new-pos [x y size xstart ystart]
  (cond
    (and (= 3 x) (= 3 y))
    {:size size :xstart (+ xstart (/ size 8)) :ystart (+ ystart (/ size 8))}
    (and (= 3 x) (= 2 y))
    {:size size :xstart (+ xstart (/ size 8)) :ystart ystart}
    (and (= 3 x) (= 1 y))
    {:size size :xstart (+ xstart (/ size 8)) :ystart (- ystart (/ size 8))}
    (and (= 2 x) (= 3 y))
    {:size size :xstart xstart :ystart (+ ystart (/ size 8))}
    (and (= 2 x) (= 2 y))
    {:size size :xstart xstart :ystart ystart}
    (and (= 2 x) (= 1 y))
    {:size size :xstart xstart :ystart (- ystart (/ size 8))}
    (and (= 1 x) (= 3 y))
    {:size size :xstart (- xstart (/ size 8)) :ystart (+ ystart (/ size 8))}
    (and (= 1 x) (= 2 y))
    {:size size :xstart (- xstart (/ size 8)) :ystart ystart}
    (and (= 1 x) (= 1 y))
    {:size size :xstart (- xstart (/ size 8)) :ystart (- ystart (/ size 8))}
    ))

(defn set-new-pos2 [key size xstart ystart]
  (cond
    (= key KeyEvent/VK_UP)
    {:size (/ size 2) :xstart (+ xstart (/ size 4)) :ystart (+ ystart (/ size 4))}
    (= key KeyEvent/VK_DOWN)
    {:size (* size 2) :xstart (- xstart (/ size 4)) :ystart (- ystart (/ size 4))}
    :default {:size size :xstart xstart :ystart ystart}
    ))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; frame ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defn -main
  [& args]
  (core/invoke-later
   (-> (core/frame :id :f
                   :title "Fractal: Mandelbrot"
                   :size [640 :by 640]
                   :on-close :exit
                   :listen [:key-pressed
                            (fn [e]
                              (let [key (.getKeyCode e)
                                    pos @refpos
                                    size (:size pos)
                                    xstart (:xstart pos)
                                    ystart (:ystart pos)
                                    new-pos (set-new-pos2 key size xstart ystart)]
                                (dosync
                                 (ref-set refpos new-pos)
                                 (redraw @refpos))
                                (core/config!
                                 (core/select (core/to-root e) [:#label]) :icon (img))
                                (core/repaint!
                                 (core/select (core/to-root e) [:#label])))
                              )]
                   :content
                   (core/label
                    :id :label
                    :icon (ImageIcon. (img))
                    :listen [:mouse-clicked
                             (fn [e]
                               (let [x (position (.getX e))
                                     y (position (.getY e))
                                     pos @refpos
                                     size (:size pos)
                                     xstart (:xstart pos)
                                     ystart (:ystart pos)
                                     new-pos (set-new-pos x y size xstart ystart)]
                                 (dosync
                                  (ref-set refpos new-pos)
                                  (redraw @refpos))
                                 (core/config!
                                  (core/select (core/to-root e) [:#label]) :icon (img))
                                 (core/repaint!
                                  (core/select (core/to-root e) [:#label])))))]))
        core/pack!
```

```
          core/show!)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```