Chapter 5 (NUPSC 2018 講習資料)

Chapter 5 は基本的に Chapter 1 から Chapter 4 まで学んできた内容に基づいた応用的な練習です。 そしていくつかの ClojureScript の便利機能も紹介します。 競技会での設問は基本的に Chapter 1 から Chapter 4 で触ふれた応用な質問となりますので、Chapter 5 はそのためのよい準備になるかも知れません。

演習5.1a

ClojureScript では `def` があることは以前紹介しました。 しかし実際 `def` は基本的に「グローバル」な名前を作ることにしか使われません。 グローバルな名前というのは全てのコードの箇所(勿論 `def` が使われた後でしか効力ありませんが)で使える名前です。 ClojureScript ではその「ローカル」バージョンも用意されています。 それは `let` というものです。 まずは例をみてみましょう

```
(let [x 10
      y 20]
  (+ x y))
```

```
30
```

このプログラムのでは、xとyは `let` の式の中でしか10と20の名前になっていません。一般的に `let` はこのように使われています。 ちなみに *名前1などが使われている計算式* はしばしば「ボディー」とも呼ばれています。

```
(let [名前1 値1
      名前2 値2
      ...]
名前1などが使われている計算式)
```

以下のコードブロックにある関数 `dist` を `let` を用いて書き換えよ。 完成させたコードブロックには `(defn dist [a b] ..)` 以外の `def` や `defn` はあってはいけない。 又 `;` で始まる行は ClojureScript では「コメント」と呼ばれていて、 プログラムを実行するとき無視されます。

```
; a, b は [1 3] のような二つの要素のあるシーケンスをもって表わされた二次元の座標である。
(defn dist [a b]
  (let [sq (fn [x] (* x x))
        x1 (first a)
        x2 (first b)
        y1 (nth a 1)
        y2 (nth b 1)]
    (Math/sqrt (+ (sq (- x1 x2))
                  (sq (- y1 y2))))))
```

```
#'cljs.user/dist
```

演習5.1b

以前ベクターに対して `(get vector index)` が紹介されていましたが、 `get` は実はベクターにだけ使用することができ、 一般的なシーケンスには使用できません。 一般的なシーケンスに `get` を使いたい場合、 `(nth sequence index)` を使えばよい。 但し、*nth*はベクターにも使いますが、 *index* が大きい場合、 *nth* は *get* より大分遅い。

演習5.1aにある `(first (rest ??))` の部分を `nth` で書き変えなさい。

演習5.1c

ClojureScript では例えば二次元座標を二要素のベクター・シーケンスで表すことがしばしばありますので、 その中身を便利に取りだす機能が実はあります。 以下の例をみてみましょう。

```
(def a [1 3])
(def b [2 4])
(let [[x1_ y1_] a]
  (def x1 x1_)
  (def y1 y1_))
(def c
  (let [[x1 y1] a
        [x2 y2] b]
    [(+ x1 x2) (+ y1 y2)]))
[x1 y1 c]
```

```
[1 3 [3 7]]
```

この機能は「構造破壊 (destructuring)」と呼ばれています。 ClojureScript では構造破壊は基本的に新しい名前を導入する全ての場所で使えます。 例えば関数の引数の宣言や `loop` 構文の変数の宣言のところも使えます。 一例をみましょう。

```
(def a [1 3])
(def b [2 4])
(defn add [[x1 y1] [x2 y2]]
  [(+ x1 x2) (+ y1 y2)])
(add a b)
```

```
[3 7]
```

構造破壊機能を使い、以下のコードブロックで平面上両点の距離を求める関数 `dist` を完成せよ。

```
(defn dist [a b] 0)
```

```
#'cljs.user/dist
```

又補足としてこの構造破壊機能はシーケンスだけではなく、 マップに対しても使えます。 例えば

```
(def person {:name  :alice
             :age   13
             :hobby :classical-music})
(let [{nm :name
       ag :age
       hb :hobby} person]
  [nm :aged ag :likes hb])
```

```
[:alice :aged 13 :likes :classical-music]
```

そして「破壊式」の最後に `:as` *全体の名前* を付け加えば破壊している値全体にも名前を付けることができる。 (注: 「破壊式」は一般的な呼びかたではなく、単に便利上ここでそう呼んでいるだけである) 例えば

```
(def person {:name  :alice
             :age   13
             :hobby :classical-music})
(let [{nm :name
       :as pn} person]
  [:we :have :a :person :named nm :and :it :is pn])
(def a [3 2])
(let [[x y :as pt] a]
  [:point pt :has :x-pos :of x :and :y-pos :of y])
```

```
[:point [3 2] :has :x-pos :of 3 :and :y-pos :of 2]
```

演習5.1d

又 ClojureScript では `for` という便利なものがあります。 これは計算機科学では一般的に「リスト内包表記」と呼ばれるものであり、 ClojureScript ではシーケンスを作ることができます。 まずは例をみましょう

```
(def xs (range 10))
(def ys (range 7))
(for [x xs
      y ys]
  [x y])
```

```
([0 0]
 [0 1]
 [0 2]
 [0 3]
 [0 4]
 [0 5]
 [0 6]
 [1 0]
 [1 1]
 [1 2]
 [1 3]
 [1 4]
 [1 5]
 [1 6]
 [2 0]
 [2 1]
 [2 2]
 [2 3]
 [2 4]
 [2 5]
 [2 6]
 [3 0]
 [3 1]
 [3 2]
 [3 3]
 [3 4]
 [3 5]
 [3 6]
 [4 0]
 [4 1]
 [4 2]
 [4 3]
 [4 4]
 [4 5]
 [4 6]
 [5 0]
 [5 1]
 [5 2]
 [5 3]
 [5 4]
 [5 5]
 [5 6]
 [6 0]
 [6 1]
 [6 2]
 [6 3]
```

```
 [6 4]
 [6 5]
 [6 6]
 [7 0]
 [7 1]
 [7 2]
 [7 3]
 [7 4]
 [7 5]
 [7 6]
 [8 0]
 [8 1]
 [8 2]
 [8 3]
 [8 4]
 [8 5]
 [8 6]
 [9 0]
 [9 1]
 [9 2]
 [9 3]
 [9 4]
 [9 5]
 [9 6])
```

`for` は `let` などと似たような名前の宣言の部分がありますが、 `for` の場合単純に値に名前を付けるのではなく、 例えば `(for [x [1 2 3]] (+ x 1))` の場合、 `[1 2 3]` の要素を一つずつ取り出し、 `x` と名付けて `(+ x 1)` という演算をします。 そして全ての演算結果をシーケンスで返します。

`for` で複数の名前・シーケンスのペアを指定するとき、 名前上のコードブロックの例にあるように、 全ての取り出しかたが枚挙され、その演算結果はシーケンスで返されます。

そして `for` では `:when` を用いて演算を行う条件を指定することもできます。 例えばこのように `(= x y)` を除外した全て$(0, 0)$から$(9, 9)$の点を枚挙することができます。

```
(def xs (range 10))
(def ys (range 10))
(for [x xs
      y ys :when (not (= x y))]
  [x y])
```

```
([0 1]
 [0 2]
 [0 3]
 [0 4]
 [0 5]
 [0 6]
```

```
[0 7]
[0 8]
[0 9]
[1 0]
[1 2]
[1 3]
[1 4]
[1 5]
[1 6]
[1 7]
[1 8]
[1 9]
[2 0]
[2 1]
[2 3]
[2 4]
[2 5]
[2 6]
[2 7]
[2 8]
[2 9]
[3 0]
[3 1]
[3 2]
[3 4]
[3 5]
[3 6]
[3 7]
[3 8]
[3 9]
[4 0]
[4 1]
[4 2]
[4 3]
[4 5]
[4 6]
[4 7]
[4 8]
[4 9]
[5 0]
[5 1]
[5 2]
[5 3]
[5 4]
[5 6]
[5 7]
[5 8]
[5 9]
[6 0]
[6 1]
[6 2]
[6 3]
```

```
 [6 4]
 [6 5]
 [6 7]
 [6 8]
 [6 9]
 [7 0]
 [7 1]
 [7 2]
 [7 3]
 [7 4]
 [7 5]
 [7 6]
 [7 8]
 [7 9]
 [8 0]
 [8 1]
 [8 2]
 [8 3]
 [8 4]
 [8 5]
 [8 6]
 [8 7]
 [8 9]
 [9 0]
 [9 1]
 [9 2]
 [9 3]
 [9 4]
 [9 5]
 [9 6]
 [9 7]
 [9 8])
```

`for` と構造破壊を用いて演習4.2dを以下のコードブロックで書き直せ。

```
(defn sum-edge-labels [g]
  (reduce + (map #(get % 2) (get-edges g))))
(defn dist [a b]
  (let [sq (fn [x] (* x x))
        [x1 y1] a
        [x2 y2] b]
    (Math/sqrt (+ (sq (- x1 x2))
                  (sq (- y1 y2))))))
(defn enum-all-edges [points]
  (for [a points
        b points
        :when (not= a b)]
    [a b (dist a b)]))
(defn points-comp-graph [points]
  (reduce (fn [g [a b weight]]
            (add-edge g a b weight))
          (empty-graph)
          (enum-all-edges points)))
(defn mst-cost [points]
  (sum-edge-labels
    (minimal-spanning-tree
      (points-comp-graph points)
      identity)))
```

```
#'cljs.user/mst-cost
```

演習5.2a

Chapter 3では我々は状態遷移関数についてみてきました。 但しChapter 3では我々が扱ったのは固定されているような状態遷移関数ばかりでした。 単一の固定された状態遷移関数でもいろいろな現実問題の数理モデルになり得ますが、 より多い現実問題では変動する条件にしたがい、状態遷移関数を調整する必要がしばしばあります。その時その変動する条件を「コマンド」だと考えてそれぞれのコマンドに対応している状態遷移関数を生成する関数を用意すれば対応できます。

また時計を例として使いましょう。 今度は我々は秒針の状態だけではなく、 分針と秒針両方扱うことにしよう。 となりますと、状態は`{:min 7 :sec 10}`のようなマップとして扱うのが適切であろう。

ClojureScript ではベクターはコマンドを表すにふさわしい。 我々の時計シミュレーションシステムでは例えばコマンドとして対応するのは `[:advance-sec` *秒針が進む秒数*`]` `[:advace-min` *分針が進む分数*`]` のようなものとしましょう。

以下のプログラムを完成させ、 `compile-watch-cmd` を上記のようなコマンドを対応している状態遷移関数に変換する関数にせよ。 但し秒針と分針の目盛は０～５９とする。

```
(defn fadvance [prev key x]
  (assoc prev key
    (mod (+ (prev key) x) 60)))
(defn compile-watch-cmd [cmd]
  (let [[c x] cmd]
    (if (= c :advance-sec)
      (fn [prev]
        (fadvance prev :sec x))
      (fn [prev]
        (fadvance prev :min x)))))
```

```
#'cljs.user/compile-watch-cmd
```

演習5.2b

通常コマンドは単発ではなく、 複数のコマンドを連続的に処理しないといけない。 以下のプログラムを完成させ、 `compile-watch-cmds` を演習5.2aにあるようなコマンドのシーケンスcmdsと初期状態prevを受け取り、 cmdsの全てのコマンドがprevに実行された最終状態を返す関数にせよ。

```
(defn fadvance [prev key x]
  (assoc prev key
    (mod (+ (prev key) x) 60)))
(defn compile-watch-cmd [cmd]
  (let [[c x] cmd]
    (if (= c :advance-sec)
      (fn [prev]
        (fadvance prev :sec x))
      (fn [prev]
        (fadvance prev :min x)))))
(defn compile-watch-cmds [cmds prev]
  (reduce (fn [prev cmd]
            ((compile-watch-cmd cmd) prev))
          prev cmds))
```

```
#'cljs.user/compile-watch-cmds
```