CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 12 Example, Assignment 3
Oct 14, 2014

Copyright ©, All rights reserved. 2014 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/openpub/) license defines the copyright on this document.
Battleship Example
The Problem

Context - Writing a battleship game

Need a function that determines
Is an enemy ship within range of our ships weapon
But weapon has a blast area so cannot use weapon if
   Enemy ship is too close to us or other friendly ships
First Pass

Assume we are at origin  Point - [x y]
Given a point & range
Is point within range

(defn in-range-1
  [position range]
  (let [pos-x (first position)
        pos-y (last position)
        target-distance (Math/sqrt (+ (* pos-x pos-x) (* pos-y pos-y)))]
    (< target-distance range)))

(in-range-1 [1 1] 1)  false
(in-range-1 [1 1] 2)  true
Second Pass

Let our position be any location

(defn in-range-2
  [position own-position range]
  (let [pos-x (first position)
        pos-y (last position)
        own-x (first own-position)
        own-y (last own-position)
        dx (- pos-x own-x)
        dy (- pos-y own-y)
        target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))
    (< target-distance range)))

This is a Java program using Clojure syntax
Second Pass - a

Using destructuring

What do we gain? lose?

(defn in-range-2a
  [[pos-x pos-y] [own-pos-x own-pos-y] range]
  (let [dx (- own-pos-x pos-x)
         dy (- own-pos-y pos-y)
         target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (< target-distance range)))
Second Pass - b

With map

What do we gain? lose?

(defn in-range-2b
 [position own-position range]
 (let [[dx dy] (map - position own-position)
       target-distance (Math/sqrt (+ (* dx dx) (* dy dy)))]
   (< target-distance range)))
Second Pass - c

Using map & reduce

What do we gain? lose?

(defn in-range-2c
  [position own-position range]
  (let [delta (map - position own-position)
      target-distance (Math/sqrt (reduce + (map * delta delta)))]
    (< target-distance range)))
(defn in-range-3
  [safe-distance range own-position position friend-position]
  (let [delta (map - position own-position)
        target-distance (Math/sqrt (reduce + (map * delta delta)))
        friend-delta (map - position friend-position)
        target->friend (Math/sqrt (reduce + (map * friend-delta friend-delta)))
    (and
     (< safe-distance target->friend)
     (< safe-distance target-distance range)))))
(defn distance-between
[a b]
(let [delta (map - a b)]
  (Math/sqrt (reduce + (map * delta delta))))

(defn in-range-3a
[safe-distance range self target friend]
(and
  (< safe-distance (distance-between friend target))
  (< safe-distance (distance-between self target) range)))

(def in-torpedo-range (partial in-range-3a 1.5 20))
(def in-cannon-range (partial in-range-3a 3 500))
What is the Abstraction?

What are we doing?

Dealing with circles shapes

Union
Intersection
Complement

Is a point in a shape
circle - returns a function

(defn circle
  ([radius]
   (circle [0 0] radius))
  ([center radius]
   (fn
    [point]
    (<= (distance-between center point) radius))))

(def small-circle (circle 1))

(small-circle [0.5 0])  true
(small-circle [1 2])    false
outside

(defn outside
[shape]
(complement shape))

(def small-circle (circle 1))

((outside small-circle) [0.5 0]) false
((outside small-circle) [1 2]) true
(defn union
  ([shape]
    shape)
  ([shape-a shape-b]
    (fn [point]
      (or (shape-a point) (shape-b point)))))
  ([shape-a shape-b & shapes]
    (fn [point]
      (let [all-shapes (conj shapes shape-a shape-b)]
        (reduce #(or %1 (%2 point)) false all-shapes)))))
Higher Level in range

(defn in-range-4
  [safe-distance range self target friend]
  (let [self-safe-zone (outside (circle self safe-distance))
        friend-safe-zone (outside (circle friend safe-distance))
        weapon-area (circle self range)
        target-zone (intersection weapon-area friend-safe-zone self-safe-zone)]
   (target-zone target)))
Assignment 3
(def sdsu-roman-numeral
  (partial clojure.pprint/cl-format nil "~@R"))
(defn sdsu-rotate [n lst]
  (if (neg? n)
      (sdsu-rotate-helper (* n -1) (reverse lst) true)
      (sdsu-rotate-helper n lst false)))

(defn sdsu-rotate-helper [n lst rev]
  (if (list? lst)
      (sdsu-rotate-helper n (vec lst) rev)
      (if (zero? n)
          (if rev
              (vec (reverse lst))
              lst)
          (sdsu-rotate-helper (dec n) (conj (subvec lst 1) (first lst)) rev))))
(require 'clojure.set :refer [union])

(defn sdsu-sum [num01 num02 maxMultiple]
  (reduce + (union (set (multiplesOfXUnderMax num01 maxMultiple))
                  (set (multiplesOfXUnderMax num02 maxMultiple)))))

(defn multiples [resultMultiples n currMultiple maxMultiple]
  (let [currResult (* n currMultiple)]
    (if (or (>= currResult maxMultiple) (< currResult 0) (>= currMultiple maxMultiple))
      resultMultiples
      (multiples (cons currResult resultMultiples) n (inc currMultiple) maxMultiple))))

(defn multiplesOfXUnderMax [x maxMultiple]
  (if (or (< x 0) (< maxMultiple x))
    (list 0)
    (multiples (list x) x x maxMultiple)))
(defn find-hundreds-place [number]
  (cond
    (= (first number) \1) "C"
    (= (first number) \2) "CC"
    (= (first number) \3) "CCC"
    (= (first number) \4) "CD"
    (= (first number) \5) "D"
    (= (first number) \6) "DC"
    (= (first number) \7) "DCC"
    (= (first number) \8) "DCCC"
    (= (first number) \9) "CM")
  )

(defn find-hundreds-place [number]
  (condp = (first number)
    \1 "C"
    \2 "CC"
    \3 "CCC"
    \4 "CD"
    \5 "D"
    \6 "DC"
    \7 "DCC"
    \8 "DCCC"
    \9 "CM")
  )
(def replace-chars


}

)

)

(defn sdsu-dna-count [dna]

(let [str-dna (replace replace-chars dna)]
  (frequencies str-dna)
)

)
(defn sdsu-palindrome
  "Higher order function calling palindrome function by passing palindrome-value into it."
[value]
(cond
  (> value 1)
    (last (sort (filter (complement nil?)
                   (into [] (palindrome value))))))
:else "Please enter number greater than 1")
Some Solutions
rotate

(defn sdsu-rotate
  [n sequ]
  {:pre [(integer? n) (or (seq? sequ) (vector? sequ) (nil? sequ))]}]
  (let [sequ-len (count sequ)]
    (if (zero? sequ-len)
      sequ
      (if (neg? n)
        (sdsu-rotate (- sequ-len (mod (- n) sequ-len)) sequ)
        (concat (drop (mod n sequ-len) sequ)(take (mod n sequ-len) sequ)))))
rotate

(defn sdsu-rotate
  [n xs]
  (let [z (mod n (count xs))]
    (concat (drop z xs) (take z xs))))

(defn sdsu-rotate
  [n xs]
  (apply concat (reverse (split-at (mod n (count xs)) xs))))
Sum multiples of 3 & 5 less then 1000

(defn multiple-of-3-or-5? [n]
  (or (= 0 (mod n 3))
      (= 0 (mod n 5))))

(apply + (filter multiple-of-3-or-5? (range 1000)))

(defn multiple-of-3-or-5? [n]
  (or (zero? (rem n 3))
      (zero? (rem n 5))))

(reduce + (filter multiple-of-3-or-5? (range 1000)))
Using Lazy

(defn sdsu-sum
  [n1 n2 max]
  (reduce + (distinct (concat (range n1 max n1) (range n2 max n2)))))
Palindrome

(defn palindrome?
  [n]
  (let [string-n (str n)]
    (= (seq string-n) (reverse string-n)))))

(defn- generate-numbers
  [digits]

  (for [x (range (int (Math/pow 10 digits)) (Math/pow 10 (dec digits)) -1 )
        y (range (int (Math/pow 10 digits)) (dec x) -1 )
        (* x y))]

(defn sdsu-palindrome
  [number]
  (let [numbers (generate-numbers number)]
    (reduce max (filter palindrome? numbers))))
(defn sdsu-dna-count
  [s]
  (when (string? s)
    (into {}
      (for [[k v] (frequencies s)]
        [(keyword (str k)) v]))))
(defn sdsu-digits
  [n b]
{:pre [(integer? n) (>= n 0) (integer? b) (pos? b)]}
(if (zero? n)
  [0]
  ((fn acc
     [number base-b-representation]
     (if (zero? number)
       (vec base-b-representation)
       (acc (int (/ number b)) (conj base-b-representation (mod number b)))))) n ()))))
(defn sdsu-roman-numeral
  [n]
  {:pre [(integer? n) (< n 4000) (pos? n)]}
  ((fn acc [
      remainder ; Remaining (unrepresented) decimal part of the number
      roman-rep ; Roman numeral representation built so far
    ]
    (cond
      (>= remainder 1000) (acc (- remainder 1000) (str roman-rep "M" ))
      (>= remainder 900) (acc (- remainder 900) (str roman-rep "CM"))
      (>= remainder 500) (acc (- remainder 500) (str roman-rep "D" ))
      (>= remainder 400) (acc (- remainder 400) (str roman-rep "CD"))
      (>= remainder 100) (acc (- remainder 100) (str roman-rep "C" ))
      (>= remainder 90) (acc (- remainder 90) (str roman-rep "XC"))
      (>= remainder 50) (acc (- remainder 50) (str roman-rep "L" ))
      (>= remainder 40) (acc (- remainder 40) (str roman-rep "XL"))
      (>= remainder 10) (acc (- remainder 10) (str roman-rep "X" ))
      (>= remainder 9) (acc (- remainder 9) (str roman-rep "IX"))
      (>= remainder 5) (acc (- remainder 5) (str roman-rep "V" ))
      (>= remainder 4) (acc (- remainder 4) (str roman-rep "IV"))
      (>= remainder 1) (acc (- remainder 1) (str roman-rep "I" ))
      :else roman-rep)) n ""))