CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 8 Assignment 1 Comments
Sep 22, 2015

Some Solutions

Problem 1

```
(defn bill-total [bill]
  (reduce + (for [x bill] (* (:price x) (:quantity x)))))
```

```
(defn bill-total [bill]
  (reduce + (map (fn [x] (* (:price x) (:quantity x))) bill)))
```

Problem 2

```
(defn combine-maps [args]
  (assoc (first args) :quantity (apply + (map :quantity args))))

;; Combines two bills and consolidates duplicate items.
(defn add-to-bill [bill items]
  (into []
        (for [[name rec] (group-by :name (into bill items))](combine-maps rec))))
```

```
(defn combine-maps [maps]
  “maps - collection of maps, each map contains :quantity key
            - other keys & values the same
   Return map with :quantity the sum of all map quantities”

  (assoc (first maps) :quantity (apply + (map :quantity maps))))

;; Combines two bills and consolidates duplicate items.
(defn add-to-bill [bill items]
  (into []
        (for [[name rec] (group-by :name (into bill items))]
             (combine-maps rec))))
```

```
(defn combine-maps [maps]
  “maps - collection of maps, each map contains :quantity key
            - other keys & values the same
   Return map with :quantity the sum of all quantities”

  (let [quantity-sum (apply + (map :quantity maps))]
       (assoc (first maps) :quantity quantity-sum)))



;; Combines two bills and consolidates duplicate items.
(defn add-to-bill [bill items]
  (into []
        (for [[name rec] (group-by :name (into bill items))]
             (combine-maps rec))))
```

```
(defn combine-maps [maps]
  “maps - collection of maps, each map contains :quantity key
            - other keys & values the same
   Return map with :quantity the sum of all quantities”

  (let [quantity-sum (apply + (map :quantity maps))]
       (assoc (first maps) :quantity quantity-sum)))



;; Combines two bills and consolidates duplicate items.
(defn add-to-bill [bill items]
  (for [[name rec] (group-by :name (into bill items))]
             (combine-maps rec)))
```

```
(defn combine-maps [maps]
  “maps - collection of maps, each map contains :quantity key
            - other keys & values the same
   Return map with :quantity the sum of all quantities”

  (let [quantity-sum (apply + (map :quantity maps))]
       (assoc (first maps) :quantity quantity-sum)))



;; Combines two bills and consolidates duplicate items.
(defn add-to-bill [bill items]
   (let [all-items  (into bill items)]
        (for [[_ rec] (group-by :name all-items)]
             (combine-maps rec)))
```

```
(defn sum-quantities [maps]
  “maps - collection of maps, each map contains :quantity key
            - other keys & values the same
   Return map with :quantity the sum of all quantities”

  (let [quantity-sum (apply + (map :quantity maps))]
       (assoc (first maps) :quantity quantity-sum)))
```

```
;; Combines two bills and consolidates duplicate items.
(defn add-to-bill [bill items]
   (let [all  (into bill items)]
        (for [[_ rec] (group-by :name all)]
             (sum-quantities rec)))
```

```
(defn make-poly [p]
  (fn [n] (reduce + (for [[a b] p] (* a (Math/pow n b))))))

;; Problem 4

(defn differentiate [p]
  (vec (for [[a b] p
             :when (not (zero? b))]
         [(* a b) (dec b)])))

;; Problem 5

;; Estimates root of a polynomial p using Newton's method with initial
;; guess x and tolerance t.
(defn find-root [t p x]
  (let [x1 (- x (/ ((make-poly p) x) ((make-poly (differentiate p)) x)))]
    (if (< (Math/abs (- x1 x)) t)
        x1
        (recur t p x1))))
```

Problem 6

```
(def account (atom 100))

(defn deposit
  [a b]
  (swap! a + b))

(defn withdraw
  [a b]
  (if (< @a b) "Insufficient funds."
        (swap! a - b)))
```

```
(defn bank_account [balance_amount]
 "Function to monitor the deposit or withdrawal money from a bank account"
 (let [balance (ref balance_amount)
       deposit (fn [amount]
                 (dosync (alter balance (partial + amount))))
       withdraw (fn [amount]
                  (dosync (alter balance #(- % amount))))]
   (fn [method_name & args]
     (cond
       (= method_name :withdraw_money) (withdraw (first args))
       (= method_name :deposit_money)  (deposit (first args))
     )
    )
  )
)

(def account (bank_account 10000))
(account :withdraw_money 400)
(account :deposit_money 100)
```

Some Issues

```
(defn bill-total [list-item]
  ;;here bill-amount will store the result
  (loop [counter (count list-item)
         i 0
         bill-amount 0.0]
    (if (<= counter 0)
        bill-amount
                    ;;here, bill-amount will have result of mulitplication between price and qunatity
      (recur (dec counter) (inc i) (+ bill-amount (* (get (get bill i) :price) (get (get bill i) :quantity))))
                    )
    )
  )
```

```
(defn bill-total [list-item]
  ;;here bill-amount will store the result
  (loop [counter (count list-item)
         i 0
         bill-amount 0.0]
    (if (<= counter 0)
      bill-amount
      (recur
        (dec counter)
        (inc i)
        (+ bill-amount (* (:price (get bill i)) (:quantity (get bill i))))))))
```

What is bill?

```
(defn calcx [eqn eqn' x]
  (- x (/ (eqn x) (eqn' x))))
(defn find-root [tolerance eqn guess]
  (let [px (make-poly eqn) px' (make-poly (differentiate eqn))]
    (let [x1 (calcx px px' guess)]
      (if (<= (Math/abs (- guess x1)) tolerance)
        (format "%.6f" guess)
        (find-root tolerance eqn x1)))))
```

```
(defn calcx [eqn eqn' x]
  (- x (/ (eqn x) (eqn' x))))

(defn find-root [tolerance eqn guess]
  (let [px (make-poly eqn)
        px' (make-poly (differentiate eqn))
        x1 (calcx px px' guess)]
    (if (<= (Math/abs (- guess x1)) tolerance)
      (format "%.6f" guess)                    ;;don't return a string
      (find-root tolerance eqn x1)))))
```

```
(defn make-poly[x](fn polynomial[y] (+ (* (first(first x)) (exp y (second(first x)))) (* (first(secon
x)) (exp y (second(second x)))) (* (first(nth x 2)) (exp y (second(nth x 2)))))))
```

```
(defn make-poly
      [x]
      (fn polynomial [y]
            (+ (* (first(first x)) (exp y (second(first x))))
               (* (first(second x)) (exp y (second(second x)))))
               (* (first(nth x 2)) (exp y (second(nth x 2)))))))
```

```
;; Wrong name -1
( defn polynomial [ poly ]
  (fn [x](double(reduce + (map #(* (nth % 0)(reduce * (repeat (nth % 1) x )))poly)))))
```

```
(defn  find-root [epsilon list guess]
  (let [result-x ( - guess(/(/((poly-maker list)guess)((poly-maker(differentiate list))guess)) 1.0))])

  (if (<=(abs(- result-x guess))epsilon)

    result-x

    (find-root epsilon list result-x)
    )


)
```

```
(declare item-bill) ;;; added so could run code -1

(defn bill-total
  [bill]
  (if (> (count bill) 1)
    (+ (item-bill (peek bill)) (bill-total (pop bill)))
    (item-bill (peek bill))))
```

```
(defn item-bill
  [item]
  (* (get item :price) (get item :quantity)))

(bill-total bill)
```

```
(defn add-to-bill [bill items] (let [
                                    new-bill1 (new-item bill items)
                                    new-bill2 (merge-item bill items)
                                    ]
                                (new-bill new-bill1 new-bill2)

                                ))
```

```
(defn add-to-bill
    [bill items]
    (let [new-bill1 (new-item bill items)
          new-bill2 (merge-item bill items)]
      (new-bill new-bill1 new-bill2)))
```

```
(defn find-root[small-limit poly x0]
 (
  let [;Px stores the value of the evaluated polynomial
       Px (calculate-poly poly x0)
       ;P-x stores the value of the derivative of the polynomial evaluated with X0 value
       P-x (calculate-poly (differentiate poly) x0)
       ]

  (let [x (into[](take 10 (iterate #(calculate-xn poly %)x0)))



       ;x-range (range(count x))
       ;x0-x1
       root-guess-1 (- (x 0)(x 1))
       ; x2-x1
       root-guess-2 (- (x 1)(x 2))
       ; x3-x2
       root-guess-3 (- (x 3)(x 4))
       ; x4-x3
       root-guess-4 (- (x 4)(x 5))
       ; x5-x4
       root-guess-5 (- (x 5)(x 6))
       ]
    (x 4)




    )
  )
)
```

```
(defn make-poly2 [poly x]
  (loop [i 0 tot 0]
    (if(< i (count poly))
      (recur (inc i) (+ tot (* (get (get poly i) 0) (Math/pow x (get (get poly i) 1)))))
      tot
      )
    )
  )
```

```
(defn make-poly2 [poly x]
  (loop [i 0
         tot 0]
    (if (< i (count poly))
      (recur
        (inc i)
        (+ tot (* (get (get poly i) 0) (Math/pow x (get (get poly i) 1)))))
      tot
      )
    )
  )
```

Unit Tests

```
(deftest test-problem1
  (testing "Problem 1"
    (are [bill total]
      (= (int (bill-total bill)) (int total))
      [{:name "a" :price 1 :quantity 1}] 1
      [{:name "a" :price 10 :quantity 2.0}] 20.0
      [{:name "a" :price 10 :quantity 0}] 0
      [{:name "a" :price 2 :quantity 1}
       {:name "a" :price 3 :quantity 2}] 8)))
```

Used in Testing Problem 2

```
(defn vec->bill
  "Used to condense bill map size"
  [[name quantity]]
  {:name name :price 1 :quantity quantity})

(defn inflate-bill
  [bill-vec]
  (mapv vec->bill bill-vec))
```

```
(inflate-bill [["a" 1]])
=> [{:name "a", :price 1, :quantity 1}]
(inflate-bill [["a" 2] ["b" 5]])
=> [{:name "a", :price 1, :quantity 2} {:name "b", :price 1, :quantity 5}]
```

```
(deftest test-problem2
  (testing "Problem 2"
    (are [bill add result]
      (let [[bill-maps add-maps result-maps] (mapv inflate-bill  [bill add result])
            computed (add-to-bill bill-maps add-maps)
            correct? (and
                       (= (count computed) (count result-maps))
                       (= (set computed) (set result-maps)))]
        (when-not  correct?
          (println "computed: " computed)
          (println "correct answer: " result-maps))
        true)
      [["a" 1]]   [["b" 1]]    [["a"  1] ["b"  1]]
      [["a" 1]]   [["a" 2]]    [["a"  3]]
      [["a" 1]]   [["b" 1]]    [["a" 1] ["b" 1]]
      ;[["a" 1]]  []            [["a"  1]]
      ;[]          [["a" 1]]    [["a"  1]]
      [["a" 1] ["b" 2]]    [["a" 2] ["c" 2]]    [["a"  3] ["b" 2] ["c" 2]]
      [["a" 1] ["b" 2]]    [["a" 2] ["b" 2]]    [["a"  3] ["b" 4]])))
```

```
(deftest test-problem3
  (are [func-vec x y]
    (= (int ((make-poly func-vec) x)) y)
    [[1 1]] 2 2
    [[2 1]] 2 4
    [[2 1] [3 0]] 1 5
    [[3 2] [-3 0]] 2 9
    [[3 2] [-2 1] [5 0]] 1 6
    [[3 2] [-2 1] [5 0]] 2 13
    [[3 2] [-2 1] [5 0]] 3 26
    ;[] 0 0
    ))
```

```
(deftest test-problem4
  (are [func-vec derivative]
    (= (differentiate func-vec) derivative)
    [[1 1]] [[1 0]]
    [[2 2]] [[4 1]]
    [[2 3]] [[6 2]]
    [[2 20]] [[40 19]]
    [[3 3] [2 2] [1 1] [5 0]] [[9 2] [4 1] [1 0]]))
```

```
(defn test-abs
  [n]
  (max n (- n)))

(defn near
  ([x y]
    (near (float x) y 0.1))
  ([x y delta]
    (let [diff (- x y)]
      (< (test-abs diff) delta))))

(deftest test-problem5
  (are [delta func-vec start root]
    (near (find-root delta func-vec start) root)
     0.0001 [[1 2] [-1 0]] 10 1
     0.0001 [[1 2] [-1 0]] -10 -1
     0.0001 [[6 2] [1 1] [-1 0]] 10 0.3333
     0.0001 [[1 2] [-4 1] [4 0]] 10 2
     0.0001 [[1 3] [-1 2] [-8 1] [12 0]] -4 -3))
```

Use spaces, no tabs

```
;; good
(when something
  (something-else))

(with-out-str
  (println "Hello, ")
  (println "world!"))

;; bad - four spaces
(when something
    (something-else))

;; bad - one space
(with-out-str
 (println "Hello, ")
 (println "world!"))
```

Vertically align function (macro) arguments spanning multiple lines

```
;; good
(filter even?
        (range 1 10))

;; bad
(filter even?
  (range 1 10))
```

Use a single space indentation for function (macro) arguments when there are no arguments on the same line as the function name

```
;; good
(filter
 even?
 (range 1 10))

(or
 ala
 bala
 portokala)

;; bad - two-space indent
(filter
  even?
  (range 1 10))

(or
  ala
  bala
  portokala)
```

Vertically align let bindings and map keywords

```
;; good
(let [thing1 "some stuff"
      thing2 "other stuff"]
  {:thing1 thing1
   :thing2 thing2})

;; bad
(let [thing1 "some stuff"
  thing2 "other stuff"]
  {:thing1 thing1
  :thing2 thing2})
```

Optionally omit the new line between the function name and argument vector for defn when there is no docstring

```
;; good
(defn foo
  [x]
  (bar x))

;; good
(defn foo [x]
  (bar x))

;; bad
(defn foo
  [x] (bar x))
```

```
;; good
(defn foo [x]
  (bar x))

;; good for a small function body
(defn foo [x] (bar x))

;; good for multi-arity functions
(defn foo
  ([x] (bar x))
  ([x y]
   (if (predicate? x)
     (bar x)
     (baz x))))

;; bad
(defn foo
  [x] (if (predicate? x)
        (bar x)
        (baz x)))
```

Optionally omit the new line between the argument vector and a short function body

Indent each line of multi-line docstrings

```
;; good
(defn foo
  "Hello there. This is
  a multi-line docstring."
  []
  (bar))

;; bad
(defn foo
  "Hello there. This is
a multi-line docstring."
  []
  (bar))
```

```
;; good
(foo (bar baz) quux)

;; bad
(foo(bar baz)quux)
(foo ( bar baz ) quux)
```

If any text precedes an opening bracket((, { and [) or follows a closing bracket(), } and]), separate that text from that bracket with a space. Conversely, leave no space after an opening bracket and before following text, or after preceding text and before a closing bracket

Don't use commas between the elements of sequential collection literals

```
;; good
[1 2 3]
(1 2 3)

;; bad
[1, 2, 3]
(1, 2, 3)
```

Use empty lines between top-level forms

```
;; good
(def x ...)

(defn foo ...)

;; bad
(def x ...)
(defn foo ...)
```

An exception to the rule is the grouping of related defs together

```
;; good
(def min-rows 10)
(def max-rows 20)
(def min-cols 15)
(def max-cols 30)
```

Avoid functions longer than 10 LOC (lines of code).
Ideally, most functions will be shorter than 5 LOC

Avoid parameter lists with more than three or four positional parameters

Avoid forward references

Don't define vars inside functions

```
;; very bad
(defn foo []
  (def x 5)
  ...)
```

Prefer vec over into when you need to convert a sequence into a vector

```
;; good
(vec some-seq)

;; bad
(into [] some-seq)
```

Use when instead of (if ... (do ...)

```
;; good
(when pred
  (foo)
  (bar))

;; bad
(if pred
  (do
    (foo)
    (bar)))
```

Use when-not instead of (when (not ...) ...)

```
;; good
(when-not pred
  (foo)
  (bar))

;; bad
(when (not pred)
  (foo)
  (bar))
```

Use not= instead of (not (= ...))

```
;; good
(not= foo bar)

;; bad
(not (= foo bar))
```

Use lisp-case for function and variable names

```
;; good
(def some-var ...)
(defn some-fun ...)

;; bad
(def someVar ...)
(defn somefun ...)
(def some_fun ...)
```

The names of predicate methods (methods that return a boolean value) should end in a question mark

```
;; good
(defn palindrome? ...)

;; bad
(defn palindrome-p ...) ; Common Lisp style
(defn is-palindrome ...) ; Java style
```

Use -> instead of to in the names of conversion functions

```
;; good
(defn f->c ...)

;; not so good
(defn f-to-c ...)
```

Follow clojure.core's example for idiomatic names like pred and coll

f, g, h - function input
n - integer input usually a size
index, i - integer index
x, y - numbers
xs - sequence
m - map
s - string input
re - regular expression
coll - a collection
pred - a predicate closure
& more - variadic input
xf - xform, a transducer