CS 596 Functional Programming and Design
Fall Semester, 2014
Doc 8 Assignment 2 Comments
Sep 27, 2014

```
(defn sdsu-nth
  [list-items index]
    (loop [counter index original-list list-items]
      (if (zero? counter)
        (first original-list)
        (recur (dec counter) (rest original-list))
      )
    )
)
```

Formatting

Names

```
(defn sdsu-nth
  [coll index]
    (loop [counter index
           tail coll]
      (if (zero? counter)
        (first tail)
        (recur (dec counter) (rest tail)))))
```

Place all trailing parentheses on a single line

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

;; bad; distinct lines
(when something
  (something-else)
)
```

Arguments - Types, Roles, Names

sdsu-nth(ArrayList elements, int position)

```
(defn sdsu-nth
  [list-items index]
```

Need to provide expected type and role

Dynamically types languages - often just give expected type

Clojure & Collection arguments

- coll
- xs

```
(defn sdsu-reverse
  ([col] (sdsu-reverse col []))
  ([col1 col2]
   (if (empty? col1)
     col2
     (recur (rest col1) (cons (first col1) col2)))))
```

col1 col2

What is the difference?

```
(defn sdsu-reverse
  ([coll] (sdsu-reverse coll []))
  ([coll result]
   (if (empty? coll)
     result
     (recur (rest coll) (cons (first coll) result)))))
```

```
(defn sdsu-nth
[l n]
(loop [my-seq l down-count n]
(if (= down-count 0)
(first my-seq)
(recur (next my-seq) (- down-count 1)))))
```

```
(defn sdsu-nth
  [coll n]
  (loop [tail coll down-count n]
    (if (zero? down-count)
      (first tail)
      (recur (next tail) (dec down-count)))))
```

What is l?

Formatting

inc, dec, pos?, neg?, zero?

Use (inc x) instead of (+ x 1)

Use (pos? x) instead of (> x 0)

Use two spaces per indentation level

No hard tabs

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

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

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

```
(defn sdsu-no-dup
  [x]
  (reduce #(if(= (last %1) %2) %1 (concat %1 (list %2))) '() x))
```

Can you read & understand this?

```
(defn sdsu-no-dup
  [x]
  (let [check-dups (fn [past-xs x]
                     (if (= (last past-xs) x)
                       past-xs
                       (concat past-xs (list x))))]
    (reduce check-dups '() x)))
```

Better but ugly

Function literals - one form only

```
;; good
(fn [x]
  (println x)
  (* x 2))

;; bad (you need an explicit do form)
#(do
   (println %)
   (* % 2))
```

```
(defn- remove-dups
  [past-xs x]
  (let [last-x (last past-xs)]
    (if (= last-x x)
      past-xs
      (conj past-xs x))))

(defn sdsu-no-dup
  [xs]
  (reduce remove-dups [] xs))
```

Better function name

Don't nest function - too long

Use let to aid understanding

Use vector for efficiency

Better Version

```
(defn- same-as-last?
  [xs x]
  (= (last xs) x))

(defn- conj-no-dups
  "Add x to end of xs if x not a duplicate"
  [past-xs x]
  (if (same-as-last? past-xs x)
      past-xs
      (conj past-xs x)))

(defn sdsu-no-dup
  "Remove consectuive duplicates from xs"
  [xs]
  (reduce conj-no-dups [] xs))
```

Lot more text than orginial

But one can understand this version

Hard to read the loop arguments

```
(defn sdsu-no-dup
  [sequence]
  (loop [counter 1 dup-list sequence no-dup-list (list)]
    (if (= counter (count sequence))
      (concat no-dup-list (list (last dup-list)))
      (let [left (first dup-list)
            right (first (next dup-list))]
        (if (= left right)
          (recur (inc counter) (next dup-list) no-dup-list)
          (recur (inc counter) (next dup-list) (concat no-dup-list (list left))))))))
```

Formatting loop variables

Why do we need counter?
End when counter = (count sequence)
At same time dup-list has one element

```
(defn sdsu-no-dup
  [sequence]
  (loop [counter 1
         dup-list sequence
         no-dup-list (list)]
    (if (= counter (count sequence))
      (concat no-dup-list (list (last dup-list)))
      (let [left (first dup-list)
            right (first (next dup-list))]
        (if (= left right)
          (recur (inc counter) (next dup-list) no-dup-list)
          (recur (inc counter) (next dup-list) (concat no-dup-list (list left))))))))
```

Stop when dup-list has one element

lists are linked lists
Expensive to add at end
Need extra code to do so
Vectors make it easier

```
(defn sdsu-no-dup
  [sequence]
  (loop [dup-list sequence
         no-dup-list (list)]
    (if (= 1 (count dup-list))
      (concat no-dup-list (list (last dup-list)))
      (let [left (first dup-list)
            right (first (next dup-list))]
        (if (= left right)
          (recur (next dup-list) no-dup-list)
          (recur (next dup-list) (concat no-dup-list (list left))))))))
```

Use Vector

left, right? in a list

Not a tree

```
(defn sdsu-no-dup
  [sequence]
  (loop [dup-xs sequence
         no-dup-xs [] ]
    (if (= 1 (count dup-xs))
      (concat no-dup-xs dup-xs)
      (let [left (first dup-xs)
        right (first (next dup-xs))]
      (if (= left right)
        (recur (next dup-xs) no-dup-xs)
        (recur (next dup-xs) (conj no-dup-xs left)))))))
```

Better Names

```
(defn sdsu-no-dup
  [sequence]
  (loop [dup-xs sequence
         no-dup-xs []]
    (if (= 1 (count dup-xs))
      (concat no-dup-xs dup-xs)
      (let [first-element (first dup-xs)
        second-element (first (next dup-xs))
        rest-elements (next dup-xs)]
      (if (= first-element second-element)
        (recur rest-elements no-dup-xs)
        (recur rest-elements (conj no-dup-xs first-element)))))))
```

Formatting

```
(defn sdsu-dup
  [col]
  (mapcat #(repeat 2 %)
  col))
```

```
(defn sdsu-dup
  [coll]
  (mapcat #(repeat 2 %) coll))
```

```
(defn sdsu-nth [c x] (first (drop x c))
```

What is c?

```
(defn sdsu-nth [coll x] (first (drop x coll))
```

```
(defn r-sdsu-nth [a b]
  (cons a (lazy-seq (r-sdsu-nth b (+ b a)))))
```

What is a? b?

```
(declare helper-pack)
(defn sdsu-pack
  [input-list]
  (helper-pack (reverse input-list) nil nil))

(defn helper-pack
  [input-list output-list duplicate-list]
  (if (empty? input-list)
    output-list
    (if (= (first input-list) (first (rest input-list)))
      (helper-pack (rest input-list) output-list (cons (first input-list) duplicate-list))
      (helper-pack (rest input-list) (conj output-list (cons (first input-list) duplicate-list)) nil))))
```

```
(declare helper-pack)
(defn sdsu-pack
  [input-list]
  (helper-pack (reverse input-list) nil nil))

(defn helper-pack
  [input-list output-list duplicate-list]
  (if (empty? input-list)
    output-list
    (let [first-input (first input-list)
          rest-input (rest input-list)]
      (if (= first-input (first rest-input))
      (helper-pack rest-input output-list (cons first-input duplicate-list))
      (helper-pack rest-input (conj output-list (cons first-input duplicate-list)) nil)))))
```

Better but still needs work

```
( defn sdsu-dup [n]
  (reduce concat (#(take 2(repeat %)) n)))
```

n?

spacing

```
(defn sdsu-dup [collection]
  (reduce concat (#(take 2 (repeat %)) collection)))
```

Spaces and ( )

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

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

space before (, [{

No space after (, [, {

If text follows),], } add space

```
(def sdsu-reverse (fn [coll]
  (loop [[ r & more :as all] (seq coll)
         acc ' ()]
    ( if all (recur more (cons r acc))acc))))
```

Spacing

```
(defn r-sdsu-nth [c x] (when-not (neg? x)
                         (if (zero? x)
                           (first c)
                           (r-sdsu-nth (rest c) (- x 1)))))
```

Formatting

```
(defn r-sdsu-nth
  [coll x]
  (when-not (neg? x)
    (if (zero? x)
      (first coll)
      (r-sdsu-nth (rest coll) (dec x)))))
```

```
(defn sdsu-dup
    [n]
   (sort (concat n n)))
```

```
(sdsu-dup [2  1   0])
```

What is the output?

n?

```
(defn sdsu-nth
  [input-sequence n]
  (if (or (< n 0) (>= n (count input-sequence)))
    "ArrayIndexOutOfBoundsException"
    (if (< n (count input-sequence))
      (loop [count 0 updated-list input-sequence]
        (if (= count n)
          (first updated-list)
          (recur (inc count) (next updated-list))
          ))
      "Value not found.")))
```

Returning strings as Exceptions/Errors

```
(if (or (< n 0) (>= n (count input-sequence)))
    "ArrayIndexOutOfBoundsException"
```

Programs don't print to standard out

How does program know string is not valid return value

How does caller know what strings are error messages

pre and post conditions over checks

```
;; good
(defn foo [x]
  {:pre [(pos? x)]}
  (bar x))

;; bad
(defn foo [x]
  (if (pos? x)
    (bar x)
    (throw (IllegalArgumentException "x must be a positive number!"))))
```

Example of :post

```
(defn slope [p1 p2]
  {:pre [(not= p1 p2) (vector? p1) (vector? p2)]
   :post [(float? %)]}
  (/ (- (p2 1) (p1 1))
     (- (p2 0) (p1 0))))
```

Excerpt From: Michael Fogus Chris Houser. “The Joy of Clojure, Second Edition.”

Using Pre-conditions

```
(defn sdsu-nth
  [input-sequence n]
  {:pre [(>= n 0) (< n (count input-sequence))]}
  (if (< n (count input-sequence))
    (loop [count 0 updated-list input-sequence]
      (if (= count n)
        (first updated-list)
        (recur (inc count) (next updated-list))))
    "Value not found."))
```

Note how far the else part is from the first if statement

Reordering the if for readablity

```
(defn sdsu-nth
  [input-sequence n]
  {:pre [(>= n 0) (< n (count input-sequence))]}
  (if (>= n (count input-sequence))
    "Value not found."
    (loop [count 0 updated-list input-sequence]
      (if (= count n)
        (first updated-list)
        (recur (inc count) (next updated-list))))))
```

But first if is always false

Removing first if

```
(defn sdsu-nth
  [input-sequence n]
  {:pre [(>= n 0) (< n (count input-sequence))]}

  (loop [count 0
         updated-list input-sequence]
    (if (= count n)
      (first updated-list)
      (recur (inc count) (next updated-list)))))
```

Count down

```
(defn sdsu-nth
  [input-sequence n]
  {:pre [(>= n 0) (< n (count input-sequence))]}

  (loop [updated-n n
         updated-list input-sequence]
    (if (zero? updated-n)
      (first updated-list)
      (recur (dec updated-n) (next updated-list)))))
```

```
(declare generate)
(defn sdsu-no-dup
 [coll]

  (if(= (count coll) 0)
    "empty list"
    (generate coll '())
   )

)

(defn generate
 [coll newlist]

;; checking if newlist is empty,
 ;;(if (= (count newlist) 0)
(if (empty? newlist)

   ;;if true then will add first item from collection to newlist and calling generate function recursively with rest of data and newlist
      (generate (rest coll) (conj newlist (first coll)))

  ;; if false then will check if collection is not yet empty then will compare last item of newlist and first item of collection, if same will skip that
  ;;data and call generate function recursively
```

Formatting
Over commented

```
(declare generate)
(defn sdsu-no-dup
   [coll]
   (if (empty? coll)
    "empty list"
    (generate coll '())))

(defn generate
 [coll newlist]
 (if (empty? newlist)
    ;;if true then will add first item from collection to newlist and calling generate function recursively
with rest of data and newlist
       (generate (rest coll) (conj newlist (first coll)))

   ;; if false then will check if collection is not yet empty then will compare last item of newlist and
first item of collection, if same will skip that
   ;;data and call generate function recursively

      (if(not= (count coll) 0)
            (if (= (first newlist) (first coll))
                (generate (rest coll) newlist)

                ;; if last item of newlist is not same as first item of collection then will append that data
to new list and call generat function again
                (generate (rest coll) (cons  (first coll) newlist)))
```

```
(if (empty? newlist)
    ;;if true then will add first item from collection to newlist and calling
    ;;generate function recursively with rest of data and newlist
     (generate (rest coll) (conj newlist (first coll)))
```

```
(if (empty? newlist)
    ;;add first item from coll to newlist
    ;;call generate with rest of data and newlist
     (generate (rest coll) (conj newlist (first coll)))
```

Less wordy comments

```
(if (empty? newlist)
  (generate (rest coll) (conj newlist (first coll)))
```

Do the comments add anything?

```
(declare generate)
(defn sdsu-no-dup
   [coll]
   (if (empty? coll)
    "empty list"
    (generate coll '())))

(defn generate
 [coll newlist]
 (if (empty? newlist)
    (generate (rest coll) (conj newlist (first coll)))
    (if (not= (count coll) 0)
       (if (= (first newlist) (first coll))
          (generate (rest coll) newlist)
          (generate (rest coll) (cons  (first coll) newlist)))
        (reverse newlist))))
```

Finally we can see the code

```
(declare generate)
(defn sdsu-no-dup
   [coll]
   (if (empty? coll)
    "empty list"
    (generate coll '())))

(defn generate
 [coll newlist]
 (if (empty? newlist)
    (generate (rest coll) (conj newlist (first coll)))
    (if-not (empty? coll)
       (if (= (first newlist) (first coll))
          (generate (rest coll) newlist)
          (generate (rest coll) (cons  (first coll) newlist)))
        (reverse newlist))))
```

(if-not (empty? coll)

is easier to read

Conveys meaning better than

(if (not= (count coll) 0)

```
(defn get-rev
  [search_list,index, new_list]
```

Formatting

```
      (if (= index (- (count search_list) 1) )

          new_list

          (get-rev search_list (+ index 1) (cons  (nth search_list (+ index 1)) new_list))

       )
)
```

```
(defn get-rev
  [search-list, index, new-list]
  (if (= index (- (count search-list) 1) )
     new-list
     (get-rev search-list (inc index) (cons  (nth search-list (inc index)) new-list))))
```

Google knows every thing

```
(defn sdsu-nth [n seq1]
   [(seq? seq1)]
     (if (= 0 n)
        (first seq1)
        ((dotimes [i n])
         (next seq1))
       (next seq1)))
```

```
(defn sdsu-dup
  [a]
  (reduce #(conj %1 %2 %2 ) [] a ))
```

```
(defn sdsu-no-dup
  [a]
  (reduce #(if-not (= (last %1) %2)
             (conj %1 %2)
             %1)
          []
          a))
```

Some Problems for Practice

4 clojure problems

http://www.4clojure.com

Project Euler

https://projecteuler.net

Clojure solutions

http://clojure.roboloco.net/?page_id=381

Some Solutions

Problem 1

```
(defn sdsu-nth
  [coll n]
  (when (> n -1)
    (first (nthnext coll n))))

(defn sdsu-nth-2
  [coll n]
  (when (< n (count coll))
    (last (take (inc n) coll))))

(defn sdsu-nth-3
  [list stopvalue]
  (when (> stopvalue -1)
    (first (drop stopvalue list))))
```

Problem 1

```
(defn sdsu-nth-4
  [list n ]
  (first (keep-indexed #(when (= n %1) %2) list)))

(defn sdsu-nth-5
   [coll n]
    ((apply comp
         (cons first (repeat n rest)))coll))
```

Problem 2

```
(defn r-sdsu-nth
  [list n]
  (when (<= 0 n)
    (if (zero? n)
      (first list)
      (r-sdsu-nth (rest list) (dec n)))))
```

```
(defn r-sdsu-nth-2
  [list n]
  (cond
    (< n 0) nil
    (= n 0) (first list)
    :else (r-sdsu-nth (rest list) (dec n))))
```

Problem 3

```
(defn sdsu-reverse
  [x]
  (if(empty? x)
    []
    (conj (sdsu-reverse (rest x)) (first x))))
```

Problem 4

```
(defn sdsu-dup
  [coll]
  (mapcat #(repeat 2 %)  coll))

(defn sdsu-dup
  [coll]
  (reduce #(conj %1 %2 %2) [] coll))
```

```
(defn sdsu-no-dup
  [xs]
  (mapcat #(take 1 %) (partition-by identity xs)))

(defn sdsu-no-dup
  [xs]
  (reduce #(if(= (last %1) %2) %1 (concat %1 (list %2))) '() xs))
```

```
(defn sdsu-pack
  [xs]
  (partition-by identity xs))
```

Style

Clojure Style Guide

https://github.com/bbatsov/clojure-style-guide

Vertically align function arguments

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

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

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})
```

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

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

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

Small Function

```
;; 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)))
```

Commas & sequential collection literals

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

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

commas & line breaks - map literals readability

```
;; good
{:name "Bruce Wayne" :alter-ego "Batman"}

;; good and arguably a bit more readable
{:name "Bruce Wayne"
 :alter-ego "Batman"}
```

```
;; good and arguably more compact
{:name "Bruce Wayne", :alter-ego "Batman"}
```

Don't define vars inside functions

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

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

```
;; good
(if-not pred
  (foo))

;; bad
(if (not pred)
  (foo))
```

Use ->, ->> to avoid heavy nesting

```
;; good
(-> [1 2 3]
    reverse
    (conj 4)
    prn)

;; not as good
(prn (conj (reverse [1 2 3])
           4))

;; good
(->> (range 1 10)
     (filter even?)
     (map (partial * 2)))

;; not as good
(map (partial * 2)
     (filter even? (range 1 10)))
```