ДОМАШНА ЗАДАЧА 4

ФУНКЦИОНАЛНО ПРОГРАМИРАЊЕ

ЕДНОСТАВНИ ФУНКЦИИ ВО CLOJURE

Во рамки на оваа домашна треба да имплементирате три групи на функции и нивно автоматско тестирање.

ГРУПА 1

Во рамки на оваа група нема потреба од користење на функции од повисок ред (функции кои земаат функција како аргумент), но сепак ако имате потреба можете да ги употребите. Целта е да се привикнете на пишување на рекурзивни функции во Clojure. Дефинирајте ги следните функции:

а) `(atomic? v)`

Враќа `true` ако `v` не е од податочен тип колекција (collection), и `false` во секој друг случај (hint: искористете некој од вградените предикати во Clojure).

б) `(member? x lst)`

Враќа `true` ако `x` се наоѓа во `lst`.

в) `(my-count lst)`

Го враќа бројот на елементи во `lst` кои се наоѓаат на "нулто ниво". На пример, `(a (b c (d e)) f)` има 3 елементи на нулто ниво; `(b c (d e))` се смета за единечен елемент на нулто ниво. `(a b () d)` има 4 елементи на нулто ниво.

г) `(append lst1 lst2)`

Ги спојува `lst1` и `lst2` во единствена листа. На пример, ако `lst1` е `(:a :b :c)` и `lst2` е `(1 (2 3))`, резултатот треба да биде `(:a :b :c 1 (2 3))`.

д) `(zip lst1 lst2)`

Ги комбинира соодветните (на иста позиција) елементи на `lst1` и `lst2` во единствена листа со подлисти, каде секоја подлиста содржи пар елементи од двете влезни; треба да запрете кога било која од листите ќе остане без елементи. На пример, ако `lst1` е `(:a :b :c)` и `lst2` е `(1 (2 3) 4 5)`, резултатот треба да биде `((:a 1) (:b (2 3)) (:c 4))`.

ѓ) `(lookup key list-of-pairs)`

За даден израз `key` (може да биде листа) и листа со подлисти `list-of-pairs` каде секоја подлиста има пар елементи и е во облик `(key value)`, ја враќа вредноста `value` која соодветствува на `key`, или `nil` ако не постои таков пар. Може да претпоставите дека клучевите не се повторуваат во листата.

е) `(my-merge lst1 lst2)`

Двете влезни листи од цели броеви, обете подредени по растечки редослед, ги спојува во единствена листа која исто така е подредена по растечки редослед. На пример, ако `lst1` е `(3 7 12 19 19 25 30)` и `lst2` е `(4 7 10 12 20)`, резултатот треба да биде `(3 4 7 7 10 12 12 19 19 20 25 30)`.

ж) `(count-all lst)`

Го враќа вкупниот број на атомични елементи во `lst`, без оглед на кое ниво се наоѓаат. На пример, `(a (b c () (25) nil) ())` има 5 атомични елементи: `a`, `b`, `c`, `25`, и `nil`.

з) `(my-drop n lst)`

Ја враќа листата `lst` од која се отстранети првите `n` елементи. На пример, `(my-drop 3 '(a b c d e))` треба да врати `(d e)`. Ако `n` е еднакво или поголемо од должината на листата, треба да се врати празна листа, `()`.

ѕ) `(my-take n lst)`

Враќа листа од првите `n` елементи на `lst`. Ако `lst` има помалку од `n` елементи, резултатот е целата листа `lst` (hint: Употребете дополнителен аргумент (листа) кој ќе го користите како акумулатор за генерирање на резултатот, т.е. за лепење на изминатите елементи од `lst`. Многу веројатно е дека `reverse` функцијата ќе ви биде корисна.)

и) **(my-reverse *lst*)**

Ги превртува елементите од листата ***lst*** во обратен редослед. На пример, за листата **(1 2 (3 4))** се добива секвенцата **((3 4) 2 1)**.

ј) **(remove-duplicates *lst*)**

Ги отстранува дупликат елементите од нулто ниво на ***lst***. На пример, за **(1 2 3 1 4 1 2)**, **remove-duplicates** враќа секвенца со елементи **(1 2 3 4)**, при што редоследот е небитен.

к) **(my-flatten *list-of-lists*)**

Отстранува едно ниво на загради од влезната листа. На пример, ако ***lst*** е **(((1 1) (2 3)) ((5 7)))**, резултатот треба да биде **((1 1) (2 3) (5 7))**.

ГРУПА 2

Оваа група на функции најлесно се имплементираат со користење на функции од повисок ред, па користете ги секогаш кога ќе увидите дека тоа е можно и полезно.

а) `(buzz list-of-ints)`

Ја трансформира влезната листа од цели броеви така што секој број од листата делив со 7 и секој број кој има барем една цифра 7 го заменува со **`:buzz`**, додека сите останати елементи остануваат непроменети. (hint: Можете бројот да го кастирате како стринг со **`(str argument)`** и со **`(seq string)`** да добиете листа на карактери кој го сочинуваат стрингот. Искористете ја вградената **`map`** функција.)

б) `(divisors-of n)`

За влезниот позитивен цел број ***n***, ги враќа делителите на ***n***, различни од 1 и самиот ***n***. На пример, 12 има делители (**`2, 3, 4, 6)`**. (hint: Искористете **`mod`** и **`filter`**.)

в) `(longest list-of-strings)`

Го враќа најдолгиот стринг од влезната листа од стрингови ***`list-of-strings`***; ако постојат повеќе стрингови со максимална должина вратете го оној кој ќе го најдете прв. Претпоставете дека ***`list-of-strings`*** не е празна. (hint: Искористете **`reduce`**)

ГРУПА 3

Во оваа група треба да напишете имплементација на најчесто користените функции од повисок ред.

а) `(my-map f lst)`

Применете ја функцијата *f* врз секој елемент од листата *lst*, и вратете листа од резултати.

б) `(my-filter pred lst)`

Применете го предикатот *pred* врз секој елемент од листата *lst*, и вратете листа од елементите кои вратиле вистина за дадениот предикат.

в) `(my-reduce f value? lst)`

Ја применува дво-аргументната функција *f* врз *value?* и првиот елемент од секвенцата *lst* ако воопшто се направи повик со наведен аргумент *value?*, инаку ја применува функцијата врз првите два елементи од секвенцата; во следниот чекор функцијата се применува врз резултатот од првиот чекор и следниот (втор или трет) елемент во секвенцата и постапката продолжува додека не се изминат сите елементи од *lst*.

г) `(my-flat-map f lst)`

f мора да биде функција која враќа листа. `my-flat-map` ја применува функцијата *f* врз секој елемент од *lst*, ја израмнува резултантната листа со отстранување на едно ниво на загради и тоа го враќа како резултат.

ТЕСТИРАЊЕ НА ДЕФИНИРАНИТЕ ФУНКЦИИ

Покрај дефинирањето на самите функции во рамки на домашната задача ќе треба да дефинирате и тест случаи за проверка на точноста на секоја функција преку таканаречениот процес на компонентно тестирање (анг. *unit testing*). За таа цел во посебна датотека ќе треба да дефинирате компонентни тестови за сите ваши функции. Во продолжение е даден модел за unit testing во Clojure, кој треба да го проширите за сите ваши функции. За секоја функција дефинирајте најмалку 5 тест примери. За секој тест пример во коментар наведете објаснување зошто сте го избрале таквиот тест пример. Доколку има специјални случаи за функцијата задолжително истите да бидат опфатени со еден тест пример за секој таков случај.

```
(ns user (:use clojure.test))

(deftest test-my-reverse
  (is (= '(3 2 1) (my-reverse '(1 2 3))))
  (is (= '(5 (3 4) 2 1) (my-reverse '(1 2 (3 4) 5))))
  (is (= () (my-reverse ()))) )

(run-tests)
```

ПРИКАЧУВАЊЕ И ПРОВЕРКА НА ВАШЕТО РЕШЕНИЕ НА ДОМАШНАТА ЗАДАЧА

Како решение за домашната задача треба да креирате две датотеки: `domasna4.clj` (со изворниот код за вашите функции, при што задолжително треба да користите документациски стрингови во изворниот код кој го објаснуваат вашето решение) и `domasna4-test.clj` (со unit тестовите за вашите функции). Двете датотеки спакувајте ги во единствена архива која ќе ја именувате со вашиот број на индекс `XXXXXX.zip` (каде XXXXXX е вашиот број на индекс) и таквата архива прикачете ја на соодветниот линк на страницата на курсот за предметот. Нема да се прифаќаат решенија кај кои има повеќе од две датотеки во архивата. Домашната задача е индивидуална и плагијаторство е најстрого забрането! Казните за плагијаторство се истите како оние дефинирани во првата домашна задача. Плагијати не се дозволени ни за датотеката со тестови! Дел од задачите се веќе вклучени во материјалите поставени на курсот и за тие задачи можете да го користите дадениот код, но треба да креирате свои оригинални тестови.

За проверка на вашето домашно најпрво треба да го стартувате Clojure компајлерот преку командна линија и со помош на `(load-file "domasna4.clj")` да ги вчитате вашите функции. Доколку вчитувањето помине успешно тогаш вашите функции се синтаксички коректни. Во следниот чекор (откако успешно ќе ги вчитате вашите функции) со командата `(load-file "domasna4-test.clj")` ќе ја проверите логичката точност на вашите изворни кодови (*забелешка: потребно е соодветните датотеки да се наоѓаат во работниот директориум за Clojure*). Доколку сите тестови се успешно поминати компајлерот ќе ви го врати следниот резултат:

```
Testing user

Ran 20 tests containing 100 assertions.
0 failures, 0 errors.
{:test 20, :pass 100, :fail 0, :error 0, :type :summary}
```