Kurs języka Lua IIled

Lista zadań nr 6

Na zajęcia 9,12.04.2018

Za zadania z tej listy można uzyskać maksymalnie 10 punktów.
Styl kodu ma wpływ na ocenę jakości rozwiązania. Zadania powinny być napisane w osobnych plikach i zawierać zarówno testy podane w treści zadania jak i własne.
Najważniejszym parametrem (poza poprawnością) jest czytelność i „luowatość" rozwiązania (a więc zgodność ze stylem wynikającym z konstrukcji języka). Wydajność pełni rolę drugorzędną – chyba że w treści zadania jest podane inaczej, lub narzut rozwiązania przeczy zdrowemu rozsądkowi i idei użycia danej funkcjonalności.

Zadanie 1. (2p) Zmodyfikuj dekorator `cache (f [,maxsize])`, (wzorowany na functools.lru_cache) tak, aby umożliwiał dostęp do statystyk, czystej funkcji, oraz resetowanie cache. Cache może działać jako LRU (last recently used) lub spamiętywać pierwsze wywołania funkcji.
Podpowiedź. Opakuj domknięcie zwracane przez wcześniejszą wersję funkcji w tablicę z dodatkowymi metodami i skorzystaj z metametod aby tablica była *callable*.

```
function fib(n)
    if n < 2 then return n
    else return fib(n-1) + fib(n-2) end
end

cfib = cache( fib, 32) -- tworzymy cache o wielkości 32
for i=1,16 do
  print (cfib(i))
end

type(cfib) --> table
cfib:cacheInfo()--> {hits=28, misses=16, maxsize=32, currsize=16}
-- (powyższe wartości mogą się różnić)
cfib:embeddedFunction() == fib --> true
cfib:cacheClear()--> {hits=28, misses=16, maxsize=32, currsize=0}
```

Zadanie 2. (4p) Napisz (w formie modułu) klasę obsługującą ułamki zwykłe. Klasa powinna być inicjalizowana dwiema liczbami całkowitymi i obsługiwać operacje arytmetyczne (dodawanie, odejmowanie, unarny minus, mnożenie, dzielenie) pomiędzy dwoma ułamkami; wszystkie operacje boolowskie; metodę `tostring` oraz konkatenację z napisami (na wzór typu `number`). Zadbaj o to, aby ułamki zawsze były trzymane w skróconej formie. Klasa powinna także udostępniać funkcję konwertującą ułamek na liczbę.
(Szczegóły implementacji, w tym wygląd konstruktora oraz rezultat funkcji `tostring` nie są narzucone z góry.)

```
print ('Wynik: '..Frac(2, 3) + Frac(3, 4)) --> Wynik: 1 i 5/12
print (Frac.tofloat(Frac(2,3) * Frac(3,4))) --> 0.5
print (Frac(2,3) < Frac(3,4)) --> true
print (Frac(2, 3) == Frac(8,12)) --> true
```

(2p) Dodaj prawidłową obsługę operacji arytmetycznych jeśli jednym z argumentów będzie nie ułamek lecz liczba (całkowita lub zmiennoprzecinkowa), zastanów się nad zwracanym rezultatem. Dodaj obsługę operacji potęgowania jeśli drugim argumentem jest liczba całkowita (w przeciwnym wypadku operator powinien zgłosić błąd).

```
print (Frac(2, 3) + 2) --> 2 i 2/3
print (Frac(2, 3) + 2.5) --> ???
print (Frac(2, 3)^3) --> 8/27
```

Zadanie 3. (4p) Napisz (w formie modułu) klasę obsługującą wektory. Klasa powinna być inicjalizowana tablicą i obsługiwać operacje arytmetyczne: dodawanie i odejmowanie wektorów, mnożenie i dzielenie przez skalar, unarny minus, oraz mnożenie wektorów jako ich iloczyn skalarny. Oprócz tego przeciąż operator # aby zwracał normę wektora i funkcję `tostring` oraz konkatenację z napisami. (Szczegóły implementacji, w tym wygląd konstruktora oraz rezultat funkcji `tostring` nie są narzucone z góry.)

```
print (Vector{1,2,3} + Vector{-2,0,4}) --> (-1,2,7)
print (Vector{1,2,3} * 2) --> (2, 4, 6)
print (Vector{1,2,3} * Vector{2, 2, -1}) --> 3
print (#Vector{4, 3}) --> 5
```

(2p) Przeciąż operację sprawdzającą równość wektora, indeksowanie wektora po współrzędnej oraz iterator `ipairs` tak, aby kluczem dla każdej współrzędnej wektora był odpowiadający jej wektor bazowy.

```
print (Vector{1,2,3} * 2 == 2 * Vector{1,2,3}) --> true
print (Vector{3, 4, 5, 6}[3]) --> 5
for k, v in ipairs(Vector{2,2,3}) do print(k, '->', v) end
--> (1, 0, 0)  ->  2
--> (0, 1, 0)  ->  2
--> (0, 0, 1)  ->  3
```

Zadanie 4. (4p) Napisz moduł z klasą `CVector`, która nakłada na tablicę zachowania znane z wektora C. W szczególności powinien on być 0-indeksowany i traktować `nil` jak każdą inną wartość a nie delimiter długości. Konstruktor powinien przyjmować zwykłą tablicę lub inny `CVector`.
Moduł powinien udostępniać następujące metody: `at(i)` zwracająca dla danego indeksu element o tym indeksie (0-based); `clear()` usuwająca wszystkie elementy z wektora; `empty()` sprawdzająca pustość wektora; `erase(i [,j])` usuwająca elementy z wektora od podanej pozycji lub w podanym zakresie; `insert(i, e)` wstawiająca elementy na podaną pozycję (e może być tablicą lub wektorem); `pop_back()` zwracająca i usuwająca ostatni element wektora, `push_back(e)` wstawiająca element na koniec wektora; `size()` zwracająca liczbę elementów w wektorze. Wszystkie operacje modyfikują wektor na którym zostały wywołane.

```
local v = CVector{'a', 'd', 'e'}
local w = CVector(v) --> w = {'a', 'd', 'e'}
print (v:empty(), CVector{}:empty()) --> false   true
v:insert(1, {2, 3}) --> v = {'a', 2, 3, 'd', 'e'}
v:pop_back()   --> v = {'a', 2, 3, 'd'}
print (v:size()) --> 4
v:push_back(nil) --> v = {'a', 2, 3, 'd', nil}
print (v:size()) --> 5
```

(2p) Przeciąż dla klasy wektora operację konkatenacji (łączącej ze sobą wektory), długości, indeksowania, `tostring` oraz iterator `ipairs`.

```
local v = CVector{'a', nil, 'c'}..CVector{4, 5}
print (v) --> <a,nil,c,4,5>
print (#v) --> 5
print (v[0], v[#v-1]) --> a  5
for k, v in ipairs(v) do print(k, '->', v) end
--> 0  ->  a
--> 1  ->  nil
--> 2  ->  c
--> 3  ->  4
--> 4  ->  5
```