Kurs języka Lua IIIed

Lista zadań nr 7

Na zajęcia 26,30.04.2018

Za zadania z tej listy można uzyskać maksymalnie 14 punktów, w tym co najmniej 6 punktów musi być za zadania z C API.
Szczegółowe kryteria oceny zadań znajdują się na stronie przedmiotu.

Zadanie 1. (2p) Napisz własną, poprawioną i wygodną dla Ciebie implementację funkcji `StackDump`: kod ma być napisany w „czystym" C++, elementy stosu powinny mieć obok napisane indeksy (zarówno dodatnie jak i ujemne) oraz typy.
Wykorzystując tę funkcję prześledź stan stosu w trakcie następujących wywołań:

```
lua_pushnumber(L, 3.5);
lua_pushstring(L, "hello");
lua_pushnil(L);
lua_pushvalue(L, -2);
lua_remove(L, 1);
lua_insert(L, -2);
```

Zadanie 2. (4p) Napisz obsługę wymyślonego przez siebie pliku konfiguracyjnego zakodowanego w Lua. Pewne operacje w tym pliku powinny zależeć od zmiennych globalnych, które muszą być ustawione z poziomu C/C++. W pliku konfiguracyjnym powinno się znajdować co najmniej 10 wartości, w tym liczby całkowite i zmiennoprzecinkowe, napisy i wartości boolowskie. Wczytaj wszystkie te wartości, po czym wykonaj na części z nich jakieś operacje i zapisz z powrotem do stanu Lua modyfikując lub tworząc nowe zmienne globalne.
Zadbaj o poprawną obsługę błędów (możesz, ale nie musisz, skorzystać funkcji `error` z wykładu).
Np. dla pliku konfiguracyjnego

```
if verbose >= 10     then verbose_level = 'large'
elseif verbose  >= 5 then verbose_level = 'medium'
else                      verbose_level = 'low'
end
developer_debug_on = verbose_level == 'large'
window_height = 500
window_ratio = 0.75
```

oczekiwany schemat działania to

```
// utwórz nowy stan Lua z załadowanymi bibliotekami
// ustaw zmienną globalną verbose
// załaduj plik konfiguracyjny
// wczytaj z niego wszystkie wartości
// zmodyfikuj stan, np. dodając zmienną globalną
//   window_width = window_height * window_ratio
// pokaż, że stan naprawdę uległ modyfikacji
// zamknij Lua
```

Zadanie 3. (4p) Napisz (w formie modułu) klasę obsługującą drzewa prefiksowe. Drzewa powinny przechowywać sekwencje dowolnych typów. Zaprojektuj efektywnie strukturę węzłów. W szczególności, jeśli w węźle znajduje się tylko jeden sufiks, to można go trzymać w całości w tym węźle (zamiast tworzyć całą gałąź).
Wewnętrzna reprezentacja drzewa powinna być przed użytkownikiem ukryta. Drzewo powinno implementować następujące operacje (operacje modyfikujące drzewo powinny także (dla wygody) je zwracać):

- `add` – dodaje sekwencję do drzewa (w czasie zależnym od długości słowa),
- `find` – sprawdza (w czasie zależnym od długości słowa) czy podana sekwencja znajduje się w drzewie (zwraca `true` albo `nil`/`false`),
- `merge` – łączy dwa drzewa w efektywny sposób (szybciej niż `add` każdego z elementów drugiego drzewa),
- `size` – zwraca (w czasie stałym) liczbę przechowywanych w drzewie sekwencji,
- `capacity` – zwraca liczbę wierzchołków istniejących w drzewie.

Konstruktor powinien opcjonalnie przyjmować drzewo prefikowe lub sekwencję sekwencji.
Przeciąż operator + tak aby działał jako `merge`/`add` (do lewego drzewa) w zależności od typu drugiego argumentu oraz # żeby działał jako `size`.

```
local t = Trie.new()
local r = Trie.new{ {1,2,3,4,5}, {1,2,6,6,6 } }
print (t:size(), r:size()) --> 0   2
print (t:capacity(), r:capacity())  -->  1  5
print (r:find{1,2,3})  --> false
print (r:find{1,2,3,4,5})  --> true
t:add{'a','bb','ccc'}
t+{1,2,3}
print (#t, t:capacity())  --> 2  3
t:merge(r)
print (#t)  --> 4
print (t:find{1,2,3})  --> false
print((r+Trie.new{1,2,6,7,7,'a'}):capacity())  --> 7
```

(2p) Zaprojektuj iteratory `pairs` i `ipairs`: `pairs` powinien zwracać elementy drzewa w dowolnej kolejności (ale powinien być szybki), `ipairs` może być wolniejszy, ale zwracane przez niego pary pozycja, element powinny być posortowane leksykograficznie (tzn. w rosnącym porządku na „literach" sekwencji jakiegokolwiek typu by one nie były – wymyśl jakąś w miarę racjonalną metodę zachowania przy porównywaniu wartości różnych typów).

```
for e in pairs(t) do  print (tab.concat(e,',')) end
--> 1,2,6,6,6
--> 1,2,3,4,5
--> a,bb,ccc
--> 1,2,3
for i, e in ipairs(t) do  print (i, '->', tab.concat(e,',')) end
--> 1 -> 1,2,3
--> 2 -> 1,2,3,4,5
--> 3 -> 1,2,6,6,6
--> 4 -> a,bb,ccc
```

Zadanie 4. (2p) Zdobądź na CodinGame achievement *Lua Addict*

Zadanie 5. (2p) Napisz moduł, który pozwoli na łączenie wieloplikowych projektów Lua w jeden. Powinien on odczytywać plik bazowy poszukując wywołań funkcji `require` i tworzyć identycznie działający plik wynikowy, zawierający w sobie kod wszystkich wykorzystywanych modułów.
Wystarczy, że ograniczysz przeszukiwanie kodu do kilku najczęstszych sposobów wczytywania modułów, np.:

```
local m = require ("MyModule")
local mm = require'MyModule'
```

Funkcja ma przeszukiwać wczytane moduły rekurencyjnie (pomijając oczywiście swój moduł) i dbać aby, jeśli to tylko możliwe, ten sam moduł nie był wielokrotnie kopiowany. Zwróć uwagę na potencjalny problem konfliktu nazw zmiennych i zachowanie prawidłowej kolejności wczytywania modułów.

Zadanie 6. (4p) Przedstaw wykonane podpunkty zadań dotyczących klas z poprzedniej listy (zadania 2, 3, 4) których jeszcze nie oddałeś, za maksimum 4 punkty.