Kurs języka Lua 2017

Lista zadań nr 7

Na zajęcia 24–25.04.2017

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. 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 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. Pisząc w Lua weź udział w zawodach Coders Of The Caribbean odbywającym się na CodinGame w dniach 14.04–24.04[1] i:
(2p) zdobądź srebro,
(2p) zdobądź złoto,
(2p) zdobądź legendę.

Zadanie 4. Zdobądź na CodinGame achievement:
(2p) *Lua Lover*
(2p) *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) Napisz dekorator `typecheck (f [, retvals=1], ...)` (luźno wzorowany na Pythonowym typecheck-decorator), który będzie dynamicznie sprawdzał zgodność typów podczas wywoływania zwróconej przez niego funkcji.
Jeśli drugi argument `retval` dekoratora jest typu `integer`, oznacza on liczbę argumentów zwracanych przez tę funkcję (domyślnie 1). Następnie dekorator powinien wczytać `retval` argumentów anotujących wartości zwracane przez funkcję. Wszystkie późniejsze argumenty są anotacjami dla kolejnych argumentów dekorowanej funkcji.
Anotacja może być napisem, tablicą napisów lub `nil`. Jeśli jest tablicą, to wartość jest poprawna jeśli spełnia którykolwiek z podanych typów. Jeśli ma wartosc `nil` to jej typ nie jest w żaden sposób ograniczony. Anotacje powinny obsługiwać następujące typy:

```
'table', 'string', 'function', 'bool', 'number',
'integer', 'float', 'nil'
```

Dodatkowo jeśli napis kończy się gwiazdką, to jest skrótem od sumy z `nil`

```
'table*' --> {'table', 'nil'}
'bool*'  --> {'bool', 'nil'}
-- number jest równoważny integer or float
'number' -->  {'integer', 'float'}
```

Jeśli parametr może być napisem, powinnśmy mieć możliwość dopasowania go do wzorca, tzn. anotacja postaci

```
'string:PATTERN'
```

powinna się dopasować tylko jeśli wyrażenie `PATTERN` całkowicie opisuje podaną wartość.
Dekorator powinien informować użytkownika o błędnych wywołaniach (pamiętaj o wskazaniu odpowiedniego miejsca popełnienia błędu).
(Szczegóły implementacji, w tym wygląd komunikatu o błędzie nie są narzucone z góry.)

[1] Daty zakończenie nie jestem w 100% pewny.

```
local fun = function (x, y)
  return x+y < 10, x > 0 and {x, x+y, x+2*y} or print
end

local tcfun = typecheck(fun, 2, 'bool', 'table',
                'integer', {'number', 'string'})

tcfun(10, 20) --> OK
tcfun(10, '20.0') --> OK
tcfun(10.0, '20.0')
--> Function call error: argument 1 is 10.0 not an integer
tcfun(10.0, nil)
--> Function call error: argument 2 is nil not a number or string
tcfun(-5, 20)
--> Function call error: return value 2 is a function not a table

local tcf = typecheck(someF, 'integer',  nil, 'number*', 'string:[rgb]')
tcf({}, nil, 'r') --> OK
tcf({}, nil, 'R')
--> Function call error: argument 3 is 'R' not a string matching [rgb]
tcf(127, 23.5, 'rgb')
--> Function call error: argument 3 is 'rgb' not a string matching [rgb]
```

Zadanie 7. (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
```

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
```