1. jakies wazne terminy i pojecia

w 2 rozdziale jak jest zaimplementowana lua (to chyba polaczyc z 3 bo w 3 bedzie chyba malo)

w 3 jak ona dziala (wazne informacje powiazane z moim programem)

w 4 moze byc juz o moim programie

- program może tez cos prezentowac

Jak wiadomo, wiele rzeczy w dzisiejszych czasach zostało bardzo zautomatyzowane. Dotyczy to nie tylko zastąpienia ludzi robotami w miejscach pracy gdzie potrzebna jest bardzo duża precyzja produkcji, ale także do rutynowych, prostych (głównie schematycznych, które można opisać jakimiś algorytmami), aczkolwiek żmudnych prac. Automatyzacje mają także miejsce w systemie komputerowym: na przykład użytkownik może codziennie logować się na swoją pocztę czy wykonywać jakieś inne proste czynności, które musi powtarzać co ileś dni.

**- czym jest jezyk skryptowy**

Jeśli zapytamy programistę czym jest język skryptowy, zapewne w większości przypadków otrzymamy parę przykładów jaki język należy do tej kategorii. Jednak większość osób tak naprawdę nie wie dlaczego. Znaczna liczba źródeł definiuje go jako język umożliwiający sterowanie konkretnymi zadaniami w systemie (ang. Job Control)[[1]](#footnote-2). Wywodzi się on głównie z czasów superkomputera (ang. mainframe) firmy IBM®, który używał języka JCL (Job Control Language) do uruchamiania określonych programów w systemie bez interwencji manualnej (ang. batch processing[[2]](#footnote-3)). Języki te początkowo miały bardzo dużo ograniczeń, np. język JCL umożliwiał m.in. deklarację zmiennych/parametrów, warunkowe uruchamianie zadań, część możliwości modularności oraz parametryzowane procedury, które bardziej przypominały znane z C makra[[3]](#footnote-4).

Większość tego typu języków jest także określanych jako „Glue Languages”. Pojęcie to oznacza, że takie języki służą także do łączenia komponentów (narzędzi, modułów) niekoniecznie napisanych w języku skryptowym. Dzięki temu możliwe jest połączenie programów napisanych w różnych językach, które rozwiązują pojedyncze problemy, aby wspólnie rozwiązać większy problem.

Z powodu szybkiej i specyficznej ewolucji języków skryptowych (np. Perl, który posiada cechy języka skryptowego, systemowego jak i idei maszyny wirtualnej[[4]](#footnote-5) podobnej do Java™), konkretna definicja określająca czym naprawdę jest język skryptowy jest trudna do ustalenia. Cechy charakterystyczne takich języków, które są często podawane[[5]](#footnote-6) to m.in.:

- dynamiczne typowanie, np. w Lua:

x = "string"

x = 5

oba wyrażenia są poprawne i nie wymagają ręcznego określenia typu przez programistę)

- zazwyczaj są interpretowane zamiast kompilowane

- głównie używane do pisania krótszych i prostszych programów

- czym jest lua i jakie ma możliwości

Lua jest jednym z przenośnych języków skryptowych najczęściej wykorzystywanym w branży gier komputerowych[[6]](#footnote-7), który zbliżony jest najbardziej do języków proceduralnych. Działa na każdej maszynie, która udostępnia kompilator ANSI C, począwszy od systemów wbudowanych aż do mainframe’ów. Umożliwia między innymi rozszerzalność (dodawanie) funkcji, których implementacje egzystują w zewnętrznym kodzie napisanym np. w języku C[[7]](#footnote-8), Java[[8]](#footnote-9) lub Delphi[[9]](#footnote-10). Ponadto od roku 2003, gdy zadebiutowała wersja 5.0, działa na wirtualnej maszynie Parrot, która w odróżnieniu od większości wirtualnych maszyn (które są bazowane na stosie) bazuje na rejestrach[[10]](#footnote-11), dzięki temu zwiększona została, w niektórych przypadkach ponad dwukrotnie, wydajność w dostępie do zmiennych lokalnych. Co więcej posiada automatyczne zarządzanie pamięcią wraz ze swoim własnym garbage collector’em, który został dodany w wersji 5.1[[11]](#footnote-12) oraz który jest niezwykle istotny w grach komputerowych.

- jak dziala lua

Działanie interpretera Lua oparte jest o emitowanie instrukcji dla maszyny wirtualnej „w locie”, czyli w trakcie procesu parsowania, aczkolwiek zapewniając także pewne optymalizacje, np. utworzenie kodu (wyemitowanie instrukcji) dla wyrażeń związanych ze stałymi i zmiennymi jest opóźnione. Innymi słowy interpreter najpierw generuje prostą strukturę, w której umieszcza potrzebne mu dane na ich temat i na jej podstawie optymalizuje kod[[12]](#footnote-13).

Na początek przyjrzymy się temu w jaki sposób reprezentowane są dane w języku Lua, opiszemy strukturę przechowującą je oraz jakie korzyści i wady z niej wynikają.

Następnie przejdziemy do opisu tabel, które de facto są jedynym mechanizmem do tworzenia struktur danych wykorzystywanych w Lua[[13]](#footnote-14). Wyjaśnimy tutaj także metodę Brent'a, która została w nich wykorzystana.

W kolejnym kroku opiszemy funkcje oraz ich domknięcia. Przyjrzymy się sposobowi w jaki są one reprezentowane po kompilacji oraz jak przechowywane są potrzebne jej dane.

Na koniec zostawimy opis działania maszyny wirtualnej.

- dane w lua

Danymi w Lua są: nil, boolean, liczby (domyślnie Double, ale prosta jest kompilacja Lua używając Float lub Long, z tego względu, że niektóry urządzenia nie mają wsparcia sprzętowego dla podwójnej precyzji), string, tabele, funkcje, wątki oraz własne zdefiniowane typy przez użytkownika. Dane w Lua reprezentowane są jako pary (t,v) w oznakowanych uniach w następujący sposób[[14]](#footnote-15):

|  |  |
| --- | --- |
| typedef struct {  int t;  Value v;  } TObject; | typedef unin {  GCObject \*gc;  void \*p;  lua\_Number n;  int b;  } Value; |

gdzie wartość t oznacza jakim typem jest wartość v. Wartości Boolean (pole b) oraz liczbowe (pole n) są reprezentowane bezpośrednio w unii, za to wartości typu string, tabele, funkcje, wątki (pole gc) oraz dane zdefiniowane przez użytkownika (pole p) - jako wskaźniki. Wartość nil nie jest reprezentowana za pomocą struktury Value, ponieważ do jej identyfikacji wystarczy sama wartość t.

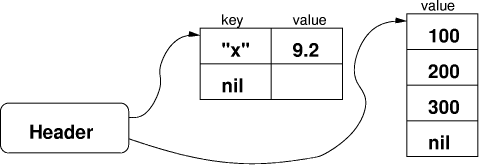
Rozwiązanie takie ma oczywiście swoje konsekwencje. Na przykład kopiowanie wartości może być w niektórych przypadkach dosyć kosztowne. Weźmy dla przykładu kopiowanie liczby 64-bitowej double na maszynie 32-bitowej:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| INT | | | | DOUBLE | | | | | | | | PADDING | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| WORD | | | | WORD | | | | WORD | | | | WORD | | | |

Rozmiar struktury TObject jest równy 12 bajtów (16 bajtów jeśli uwzględnimy tzw. padding). Maszyna 32-bitowa, kopiuje dane w porcjach po 4 bajty, tzw. słowo (ang. word), zatem aby skopiować liczbę typu double potrzebne jest 3 (lub 4) kopiowań słów.

- tabele w lua

Język ten nie udostępnia typu tablicowego, dlatego do symulowania tablic wykorzystywane są właśnie tabele. Do wersji 4.0 tabele były zaimplementowane jako zwykłe tzw. hashmapy, jednak od wersji 5.0 tabela jest połączeniem tablicy (która nie przechowuje wartości klucza) z hashmapą:



Możliwymi konfiguracjami par tutaj są x->9.2, 1->100, 2->200, 3->300. Ponadto jej rozmiar jest dynamiczny - rozszerza się gdy dodajemy do niej jakiś element oraz kurczy gdy usuwamy (poprzez przypisanie nil'a). Gdy tabela potrzebuje więcej miejsca, rozmiary tablicy liczone są na nowo:

Rozmiarem tablicy jest największe takie n, dla którego przynajmniej połowa elementów w tablicy pomiędzy 1 a n (aby uniknąć marnowania miejsca przy tablicach rzadkich) oraz przynajmniej 1 element pomiędzy a (aby uniknąć rozmiaru gdy jest wystarczające) są w użyciu. Po przeniesieniu tego na język matematyczny:

Oznaczmy tablicę jako:

Oznaczmy element tablicy, który jest w użyciu jako 1, a element, który nie jest w użyciu jako 0.

Wtedy szukamy takiego największego , że:

oraz

Dzięki temu, że tabela wykorzystuje także zwykłą tablicę, to w niektórych przypadkach możliwe jest zmniejszenie zużycia pamięci nawet dwukrotnie, ponieważ tablica nie wymaga dodatkowej pamięci na klucze. Ponadto dostęp do danych za pomocą klucza liczbowego jest szybszy, ponieważ nie wymaga wykonania procedury hashującej w połączeniu z metodą Brent'a.

- metoda brent'a

Metoda ta polega na rozwiązywaniu kolizji przy hashowaniu oraz optymalizacji dostępu do danych. Bazuje ona na koncepcji, która zakłada żeby więcej czasu poświęcić na wstawianie elementów niż ich znalezienie. Dlatego też jest ona głównie wykorzystywana przy kompilatorach lub interpreterach.

Załóżmy, że:

rozmiar\_tablicy - najbliższa liczba pierwsza, większa od liczby elementów

H1(klucz) - funkcja hashująca = klucz % rozmiar\_tablicy

H2(klucz) - przesunięcie (offset) = (klucz / rozmiar\_tablicy) % rozmiar\_tablicy

oraz

i - liczba miejsc, które musimy odwiedzić aby znaleźć element (A), który wstawiamy

j - liczba dodatkowych miejsc, które musimy odwiedzić aby znaleźć element (B), który przesuwamy

Chcemy znaleźć minimalną sumę i+j (i > 0 oraz j > 0).

Rozwiązywanie kolizji przebiega następująco:

1. Na początku musimy spróbować przesunąć element, z którym wstawiany element koliduje (oznaczmy to jako punkt ), do punktu . Gdy zmienia się w górę oznacza to, że musimy spróbować przesunąć B (aktualnie istniejący element) pod indeks przesunięty o .

2. Jeśli punkt jest zajęty to sprawdzamy czy jest możliwe zapisanie sumy w innej kombinacji (której jeszcze nie było), tzn. sumę można zapisać także jako , czyli punkt . Zmiana oznacza, że teraz naszym elementem B będzie element, który jest pod indeksem przesuniętym od B o offset . Następnie próbujemy przesunąć B o , jeśli się nie uda to próbujemy dopasować kolejne oraz . Itd.

Innymi słowy szukamy po kolei odpowiedniej minimalnej kombinacji , czyli: itd.

Jeśli jest kilka możliwych kombinacji dla przesunięć to to czy wybierzemy czy zależy od naszej implementacji, aczkolwiek głównie stosuje się tą pierwszą konwencję.

- funkcje w lua

Gdy dochodzi do momentu kompilacji funkcji, Lua generuje jej prototyp, który zawiera:

* instrukcje dla wirtualnej maszyny
* tablicę z stałymi użytymi w funkcji
* dane dla debuggera

Podczas działania programu gdy dochodzi do wywołania funkcji, silnik Lua generuje domknięcie funkcji (ang. *closure*). Każde domknięcie posiada wskaźnik do:

* prototypu od którego zostało utworzone,
* tabeli ze zmiennymi globalnymi (*environment*),
* tabeli ze zmiennymi lokalnymi z poza funkcji (*upvalues*).

- lua okiem programisty api

Jedną z najważniejszych możliwości języka Lua jest definiowanie własnych funkcji (których implementacje rezydują w naszym programie), które można wykorzystać podczas późniejszego pisania skryptów. Aby zdefiniować funkcję należy wykonać następujące kroki (kolejność dowolna):

1. Zdefiniować funkcję, która przyjmuje argument typu lua\_State\* i zwraca int.

2. Zarejestrować powyższą funkcję za pomocą void lua\_register(lua\_State\*, const char\*, lua\_CFunction);

Ad. 1. Po pierwsze, silnik Lua nie posiada globalnych zmiennych, a co za tym idzie, wszystkie wartości określające stan interpretera są trzymane w strukturze lua\_State. Dzięki temu silnik ma możliwość powrotu do określonego stanu[[15]](#footnote-16). Po drugie, funkcja musi zwracać wartość liczbową, która określa ile argumentów zwraca dana funkcja (argumenty są wrzucane do rejestrów za pomocą funkcji typu lua\_pushnumber(state, value)), ponieważ język Lua umożliwia zwracanie przez funkcje kilku wyników[[16]](#footnote-17), np.:

x,y = foo2()

Ad. 2. Tutaj należy zauważyć, że nazwa funkcji (która będzie nazwą widoczną w skryptach) może być różna od nazwy funkcji podczas deklaracji. Innymi słowy

lua\_register( luaState, "myFunction", pointerToFunction );

oznacza, że gdy w skrypcie napiszemy myFunction() to zostanie wywołana funkcja pointerToFunction(), która jest zdefiniowana w naszym programie.

Definicja funkcji zdefiniowana jest następująco:

int pointerToFunction( lua\_State \*state ) {

// pobierz argumenty z tablicy Lua i zrob cos z nimi

// zwroc informacje o tym ile argumentow zwrocila dana funkcja

return 0;

}

Interpreter języka od wersji 5.0 bazuje na rejestrach, które są umieszczane na stosie w tzw. activation records. Gdy dochodzi do momentu interpretowania zdefiniowanej przez programistę funkcji, na stosie zostaje przydzielona pamięć na activation record na tyle duża by pomieścić przebywające tam rejestry, które przechowują argumenty funkcji. Zatem aby ich użyć należy je uprzednio pobrać z rejestrów:

int \_arg1 = ( int )lua\_tonumber( state, -2 );

int \_arg2 = ( int )lua\_tonumber( state, -1 );

drugim argumentem funkcji jest indeks elementu, który chcemy pobrać z rejestru. Rejestr w tym przypadku najlepiej traktować jako zwykły stos, na który są odkładane argumenty od lewej do prawej. Dlatego kolejność "ściągania" argumentów jest odwrotna do kolejności "wrzucania". Wartość oznacza wierzchołek stosu. Im ta wartość jest mniejsza tym niższy na stosie pobieramy argument. Zauważmy, że nie musimy dbać o kolejność pobierania argumentów (w powyższym kodzie najpierw pobieramy argument, który znajduje się najniżej; domyślnie zwracamy ją jako liczbę double), ważne jest tylko podanie prawidłowego indeksu. Istnieją oczywiście inne wersje funkcji pobierającej daną z rejestru dla innych typów, np. lua\_tostring( lua\_state, int );, która zwraca const char\*.

-- wykonywanie instrukcji skryptu LuaEngine

Bardzo ważnym faktem (i oczywistym) silnika Lua jest to, że gdy skrypt zostanie wczytany, to po jego uruchomieniu silnik wykonuje każda instrukcję po kolei (w sensie tak jak napisaliśmy logikę skryptu). Jest to logiczne i wydaje się normalne zachowanie, aczkolwiek w pewnych przypadkach może to być w pewnym sensie utrudnienie.

Załóżmy, że chcemy aby nasz skrypt wykonał dwa kliknięcia myszką w obrębie jakiegoś programu. Pierwsze kliknięcie może być wciśnięciem przycisku, który wyświetla jakieś okno dialogowe, za to drugim kliknięciem chcielibyśmy wcisnąć przycisk już na oknie dialogowym. Nasz skrypt może wyglądać np. tak:

leftClickAt( 100, 100 )

leftClickAt( 200, 200 )

Powyższy skrypt zrobi dokładnie to co mu napisaliśmy (czyli kliknie dwa razy w miejscach o podanych współrzędnych), jednakże może się zdarzyć, że po pierwszym kliknięciu okno dialogowe może nie zdążyć się pojawić przed drugim kliknięciem. Rozwiązanie jest oczywiste - musimy dodać jakieś uśpienie pomiędzy tymi dwoma kliknięciami, aby okno dialogowe miało czas na wyświetlenie się.

Istnieje wiele sposobów na rozwiązanie takiego uśpienia, opiszemy część z nich wyróżniając ich wady i zalety. Będą to sposoby, które zostały zaimplementowane w programie DoForMe!.

-- nowa komenda - sleep

Pierwszym sposobem, jaki może się narzucić na myśl, jest nowa komenda, który zatrzymuje wykonanie skryptu w określonym miejscu na określony czas. Rozwiązanie takie daje szereg zalet:

* precyzja - wiemy dokładnie gdzie i na ile zostanie zatrzymane wykonanie skryptu
* modyfikowalność - bez problemu możemy dodać lub usunąć uśpienie w konkretnym miejscu
* prostota

aczkolwiek są także i wady:

* w przypadku gdy mamy dużo komend, które muszą być wywoływane w pewnych odstępach czasu, musimy dodać dużo uśpień, przez co tracimy na przejrzystości kodu
* zwiększenie objętości kodu

-- nowa opcja - odstep czasowy co kazda komende

Kolejnym pomysłem może być parametryzowalny odstęp czasowy co każdą instrukcję w taki sposób aby nie było potrzeby umieszczania co każdą komendę uśpienia. Aby osiągnąć taki efekt trzeba poczynić pewne przygotowania w przypadku wykonywania instrukcji przez silnik Lua.

Problem ten został rozwiązany za pomocą utworzenia nowego stosu (nie tego stosu, który tworzy Lua na activation records, ale własnego), na który odkładane są poszczególne komendy. Sposób wykonywania skryptu przez interpreter Lua pozostaje bez zmian, zmiana jest dopiero w przypadku implementacji poszczególnych komend. Jeszcze przed problemem "uśpienia", funkcje dostępne w skryptach miały bezpośrednio zaimplementowany kod akcji, który dana komenda miała wykonać. Zmiana dotyczy tego, że teraz ta komenda tylko wrzuca informację na stos o tym jaka funkcja ma zostać wykonywana. Gdy Lua zinterpretuje już cały skrypt to wtedy zostanie uruchomiony licznik (ang. *timer*), który co jakiś czas ściąga komendę ze stosu i ją wykonuje. Sposób ten został zaprezentowany na rysunku poniżej:

Dzięki takiemu podejściu czas pomiędzy komendami jest wartością parametryzowalną, zatem możliwa jest jego zmiana nawet podczas wykonywania skryptu. Ta właściwość zostanie wykorzystana w kolejnym rozdziale.

1. RICH MORIN, VICKI BROWN, *Scripting languages*, <http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html>, 1999 [↑](#footnote-ref-2)
2. IBM® CORPORATION, *z/OS concepts,* [http://publib.boulder.ibm.com/infocenter/ zos/basics/topic/com.ibm.zos.zconcepts/zconcepts\_book.pdf](http://publib.boulder.ibm.com/infocenter/%20zos/basics/topic/com.ibm.zos.zconcepts/zconcepts_book.pdf) str 79, 2010 [↑](#footnote-ref-3)
3. IBM® CORPORATION, *IBM System/360 Operating System Job Control Language (C28-6529-4)*, <http://www.bitsavers.org/pdf/ibm/360/os/R01-08/C28-6539-4_OS_JCL_Mar67.pdf>, 1967 [↑](#footnote-ref-4)
4. PATRICK MICHAUD, *Announce: Rakudo Star — a useful, usable, "early adopter" distribution of Perl 6*, <http://rakudo.org/2010/07/29/rakudo-star-2010-07/>, 2010 [↑](#footnote-ref-5)
5. CUNNINGHAM & CUNNINGHAM INCORPORATION, *Glue Languages,* <http://c2.com/cgi/wiki?GlueLanguages>, 2011 [↑](#footnote-ref-6)
6. GDMAG STAFF, *Front Line Award Winners – programming tools*, <http://gdmag.com/blog/2012/01/front-line-award-winners.php>, 2011 [↑](#footnote-ref-7)
7. ROBERTO IERUSALIMSCHY, *Programming in Lua - Second Edition s. 215*, 2006 [↑](#footnote-ref-8)
8. JAMES ROSEBOROUGH, IAN FARMER, *LuaJ*, <http://sourceforge.net/projects/luaj/>, 2011 [↑](#footnote-ref-9)
9. DENNIS D. SPREEN, *Lua 5.1 for Delphi* 2010 , <http://blog.spreendigital.de/2009/09/28/lua-5-1-for-delphi-2010/>, 2009 [↑](#footnote-ref-10)
10. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - The Virtual Machine*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 7, 2005 [↑](#footnote-ref-11)
11. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - Introduction*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 1, 2005 [↑](#footnote-ref-12)
12. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - An Overview of Lua's Design and Implementation*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 2, 2005 [↑](#footnote-ref-13)
13. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 – Tables,* <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 4, 2005 [↑](#footnote-ref-14)
14. W. CELES, H. L. FIGUEIREDO, R. IERUSALIMSCHY, *The Implementation of Lua 5.0 - The Representation of Values*, <http://www.jucs.org/jucs_11_7/the_implementation_of_lua/jucs_11_7_1159_1176_defigueiredo.html> sekcja 2, 2005 [↑](#footnote-ref-15)
15. ROBERTO IERUSALIMSCHY, *Reference Manul*, <http://www.lua.org/manual/5.2/manual.html#lua_State>, 2012 [↑](#footnote-ref-16)
16. ROBERTO IERUSALIMSCHY, *Multiple results*, <http://www.lua.org/pil/5.1.html>, 2012 [↑](#footnote-ref-17)