Rozdział 26

Funkcyjne podejście do Lua

Josh Chisholm

Jeśli ktoś pisze kod w popularnym języku programowania, może nigdy nie słyszał o Lua. Ale założę się, że słyszał o Angry Birds, Wikipedii lub World of Warcraft. Co więc sprawia, że ich twórcy korzystają z Lua w tych popularnych produktach? Być może to, że Lua jest lekki, międzyplatformowy, dobrze się go osadza i rozszerza, dobrze działa, ma małe zużycie pamięci i płaską krzywą uczenia się. To dobre powody. Ale chciałbym wierzyć, że *niektórych* deweloperów Lua przyciąga dlatego, że obsługuje różne paradygmaty i obejmuje różne, całkiem przyjemne możliwości programowania funkcyjnego.

Prawdopodobnie znacie trochę JavaScript. Mimo że Lua ma całkiem inne pochodzenie, wiele idei projektowych ma wspólnych z JavaScript i wydaje się do niej podobna. Z punktu widzenia składni Lua jest trochę mniej kolczasta:

```
function hello() {
    say("Hi there, I'm JavaScript");
}

function hello()
    say "Howdy, I'm Lua"
end
```

Tak jak JavaScript, Lua dużo zawdzięcza Scheme, dialektowi Lispa. Ponieważ jednak zarówno JavaScript, jak i Lua wydają się być bliżej Java i C, nie rozpoznalibyśmy szybko śladów Lisp wewnątrz nich. Gdy spojrzymy jednak na składnię, Lua ma pewne cechy, które sprawiają, że jest świetna do używania technik funkcyjnych jako część podejścia wieloparadygmatowego.

Funkcje pierwszoklasowe w Lua

Jeśli ktoś doszedł tak daleko w tej książce, wie, że powiedzenie, że funkcje są pierwszoklasowe, wskazuje tylko, że nie ma w nich nic specjalnego, więc

mogą być traktowane jak każda inna wartość. I wiemy, że funkcje wyższego rzędu robią przynajmniej jedną z dwóch rzeczy: albo przyjmują inne funkcje jako argumenty, albo zwracają funkcje. Są to ważne możliwości w językach funkcyjnych, zobaczmy więc, jak są one implementowane w Lua.

Przypuśćmy, że mamy podręczną listę kotów. Możemy wykorzystać koncepcję *tabeli* w Lua – kolekcję w stylu armii szwajcarskiej, która zachowuje się zarówno jak tablica, jak i skrót:

```
local cats = {
   { name = "meg", breed = "persian" },
   { name = "mog", breed = "siamese" }
}
```

Funkcja, która zbiera nazwy wszystkich naszych kotów, wygląda tak:

```
function namesOf (things)
    local names = {}
    for index, thing in pairs(things) do
        names[index] = thing.name
    end
    return names
end

print(namesOf(cats)[1]) --> meg
print(namesOf(cats)[2]) --> mog
```

Lua wykorzystuje indeksy (zaczynające się od 1), które są proste teoretycznie, ale można też łatwo coś popsuć.

Możemy napisać bardzo podobną funkcję `breedsOf(cats)`, lecz musielibyśmy powtórzyć kod, aby iterować przez kolekcję rzeczy. Moglibyśmy także użyć dwukrotnie składni języka o specjalnym przeznaczeniu (`for...in`). Funkcje wyższego rzędu dają nam jednolitą metodę łączenia funkcji w celu zredukowania duplikacji i możemy to wykorzystać w Lua:

```
function map (things, fn)
    local mapped = {}
    for index, thing in pairs(things) do
        mapped[index] = fn(thing)
    end
    return mapped
end

function namesOf (things)
    return map(things, function(thing)
        return thing.name
    end)
end

function breedsOf (things)
```

```
    return map(things, function(thing)
        return thing.breed

    end)
end

print(namesOf(cats)[1]) --> meg
print(breedsOf(cats)[2]) --> siamese
```

Dzięki takiemu rodzajowi ponownego wykorzystania programy funkcyjne w Lua zazwyczaj składają się z dużej liczby bardzo małych elementów.

Rekurencja w Lua

Funkcje rekurencyjne wywołują same siebie, jak poniżej:

```
function factorial (n)
    if n == 0 then
        return 1
    else
        return n * factorial(n - 1)
    end
end

print(factorial(5)) --> 120
```

Łatwo to zrozumieć, lecz wykonanie tej funkcji dla pewnych wartości n będzie powodować błąd przepełnienia stosu. Dzieje się tak, ponieważ przy każdym kolejnym uruchomieniu `factorial(n - 1)` maszyna musi zapamiętać na swoim stosie odwołanie do poprzedniej iteracji. Stos ma skończoną wielkość, więc ostatecznie kończy się miejsce i program wylatuje w powietrze:

```
print(factorial(-1))
lua: factorial.lua:5: stack overflow
stack traceback:
    factorial.lua:5: in function 'factorial'
    factorial.lua:5: in function 'factorial'
    ...
```

Lua 5.0 dodała właściwe wywołania typu tail, rodzaj funkcyjnego `goto`, pozwalające na strukturę funkcji rekurencyjnej, w której nigdy nie kończy się miejsce na stosie:

```
function factorial (n)
    return factorialUntil(n, 1)
end

function factorialUntil (n, answer)
    if n == 0 then
        return answer
    else
        return factorialUntil(n - 1, n * answer)
```