Limbajul de programare Ruby

Cristian Țicu

Ruby este un limbaj de programare de nivel înalt, interpretat dezvoltat de către Yukihiro Matsumoto în anul 1995 în Japonia. Este descris ca fiind „cel mai bun prieten” al programatorului, deoarece calitatea interacțiunii dintre acesta si codul scris a fost pusă pe primul loc prin asigurarea unei sintaxe ușoare lipsite de „zgomot vizual” și surprize (eu am avut parte însă de câteva surprize). Ruby promovează programarea naturală și plăcută fiind considerat „optimizat pentru oameni”. Sintaxa este destul de asemănătoare cu cea a limbajului Python și Perl. Ca stil de programare este un limbaj multi-paradigmă în care se folosește foarte extensiv stilul orientat obiect împreună cu un set mai restrâns de principii funcționale.

Ruby a fost influențat de către Smalltalk, Perl și Lisp. Sistemele principale ale limbajului sunt inspirate din Smalltalk, un limbaj de programare pur orientat obiect, prin urmare absolut orice în Ruby este un obiect. Lisp și Perl au influențat stilul simplist, asemănător „scripting”-ului și utilitatea practică prin puține linii de cod. Numele a fost ales de către Matsumoto într-o sesiune online de chat, opțiunile fiind Coral și Ruby. În final, Ruby a fost ales pentru că rubinul este piatra prețioasă asociată zilei de naștere a unui coleg de al său.

Despre Ruby

Ruby este complet orientat obiect: se lucrează în totalitate doar cu obiecte. Până și primitivele (de exemplu numerele) sunt obiecte cu proprietăți și acțiuni. Acest lucru aduce avantajul de a raționa cu simplitate tot ce se întâmplă în limbaj: aceleași reguli se aplică peste tot. Un exemplu de pe pagina oficială Ruby, secțiunea despre limbaj:



În Ruby (ca în multe alte limbaje ce adoptă paradigma orientată obiect) proprietățile obiectelor sunt *variabilele* atribuite instanțelor iar acțiunile se numesc *metode*. Din pricina faptului că orice este un obiect, în Ruby nu exista funcții propriu-zise, tipul primitiv de funcție lipsind in totalitate. Dacă dorim să urmăm pașii similari altor limbaje pentru a declara o „funcție”, putem scrie:



Însă de fapt Ruby a atașat o nouă *metodă* privatăunui obiect global implicit. Ca atare, aceasta nu poate fi referită ca valoare și pasată ca argument precum în limbajul JavaScript, de exemplu, nefiind considerată *first-class function*, un aspect foarte important în programarea funcțională.



Așa cum am menționat, Ruby este complet orientat obiect. Acest lucru înseamnă că oferă toate funcționalitățile cunoscute ale paradigmei, precum clase și instanțe, încapsulare, moștenire și polimorfism prin suprascrierea metodelor și utilizarea mixin-urilor prin includerea modulelor în clase. În acest proiect am ales, însă, să mă concentrez asupra paradigmei funcționale prezente în Ruby, care s-a dovedit a fi mult mai interesantă (și diabolică, uneori) decât în alte limbaje multi-paradigma, precum JavaScript sau Python. Este foarte interesant de analizat cum un limbaj care este supus complet și în totalitate regulilor paradigmei orientată obiect reușește să ofere și facilități funcționale, neschimbând regulile.

Prezența unui set mai restrâns (și destul de încâlcit) de facilități funcționale, cum ar fi *closures*, *first-class functions* și *higher-order functions* asigură etichetarea limbajului Ruby ca limbaj de programare multi-paradigmă.

Programare funcțională în Ruby

Un *closure* este definit ca o funcție împreuna cu contextul în care aceasta a fost creată, adică starea programului exterioară ei, cunoscută ca și contextul lexical. Este important de înțeles că nu contează locul unde a fost apelată efectiv funcția, ci unde a fost ea definită. Într-un *closure*, contextul din jurul funcției este ținut minte chiar și după ce funcția propriu-zisă și-a terminat execuția și este folosit la fiecare apel ulterior al funcției, fiind salvate eventualele modificări din apelurile precedente. Un exemplu din limbajul JavaScript:



JavaScript este cunoscut pentru facilitățile funcționale foarte puternice și încurajate. Nu este un limbaj funcțional pur, ci unul multi-paradigmă care promovează programarea funcțională în librăria sa standard, prin *callbacks* și *event handlers*. În JavaScript funcțiile sunt *first-class citizens*, adică pot fi stocate în variabile, pasate ca argumente, returnate din alte funcții și au propriul lor tip primitiv de date. Aceste facilități sunt folosite extensiv în librării și framework-uri moderne de frontend, cum ar fi React.

În Ruby, un closure se poate construi astfel:



Acest exemplu dovedește în același timp și existenta funcțiilor de tip first-class si higher-order: funcțiile pot fi stocate in variabile (exemplu1 și exemplu2). O funcție higher-order este definită ca o funcție care accepta o altă funcție ca parametru sau returnează o altă funcție. Acest lucru este dovedit prin createClosure care este higher-order deoarece returnează o altă funcție, deci Ruby oferă suport pentru programare funcțională.

Se poate observa o diferență în apelul funcției din closure: se folosește metoda „.call()”, în loc de un apel direct. Acesta este prețul pe care Ruby îl plătește pentru a asigura paradigma orientată obiect. Dacă am încerca să apelăm funcția în mod obișnuit:



Acest apel este tratat ca apelul unei metode „exemplu1” pe obiectul global. Deși exemplul dovedește că Ruby oferă suport pentru closures, first-class functions și higher-order functions, pasul suplimentar de a folosi „.call()” îngreunează lizibilitatea.

În exemplul de mai sus, funcția din closure este o funcție *lambda*, o funcție fără un simbol (sau nume) asociat. O trăsătură distinctivă a limbajului este posibilitatea de a defini funcții în trei moduri diferite: prin *lambda*-uri, *procs* sau *blocks*.

Trei moduri de a face...același lucru

Un *block* în Ruby nu este un obiect propriu-zis ci doar o structură sintactică prin care se poate transmite în mod implicit o bucată de cod. Poate primi argumente și poate returna o valoare, însă nu poate exista singur; este obligatoriu ca un block să fie pasat unei metode sau unei funcții. De asemenea, un comportament surprinzător este că dacă un block returnează, se va termina si funcția care dat controlul către block.



Un singur block poate fi atribuit unei funcții în acest mod și nu poate fi stocat sau pasat explicit decât daca este transformat într-un *proc* prin sintaxa &block în antetul funcției.



Un *proc* este un obiect ce conține cod, foarte asemănător unui block. Îndeplinește aceleași funcții și se poate folosi identic, doar că acesta poate exista fără să fie asociat direct unei funcții. Aceeași surpriză este prezentă și în cazul proc-urilor: dacă un proc returnează, se încheie și execuția funcției apelante. Se poate inițializa prin cuvântul rezervat proc.

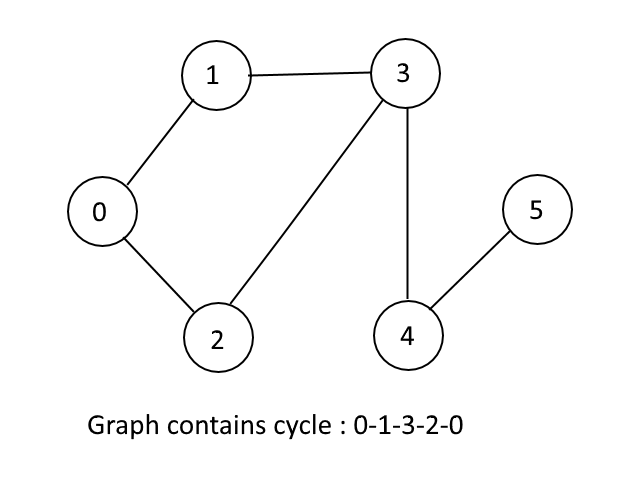


În final, avem lambda-urile, care se diferențiază de proc-uri si block-uri prin două caracteristici: un return în lambda nu oprește și funcția apelantă și aruncă eroare în cazul in care aritatea nu este respectată la apelul lamba-ului. Proc-urile și block-urile nu verifică aritatea, parametrii pasați în plus fiind ignorați iar cei absenți înlocuiți cu nil.



Problema aleasă

Problema pe care am ales-o este detectarea de cicluri într-un graf neorientat. Este o problemă importantă deoarece apare în multe domenii în care relațiile dintre entități sunt importante, cum ar fi rezolvarea dependențelor între module în cadrul compilatoarelor sau managerelor de pachete.



Am rezolvat problema folosind două paradigme: orientată obiect și funcțională. O metodă orientată obiect ar presupune folosirea unor clase cu metode și variabile de instanță și o gândire imperativă asupra codului.



O viziune funcțională, în schimb, s-ar concentra pe evitarea efectelor secundare (mutații). Asta implică crearea de obiecte noi în locul modificării lor. Un avantaj al metodei funcționale este utilizarea de higher-order functions pentru a ușura logica programului (în cazul nostru, funcțiile „any?” și „map”).



Nu am descoperit avantaje sau dezavantaje majore în oricare dintre metode, cu excepția faptului că metoda funcțională este mai ușor de testat datorită evitării efectelor secundare în funcții. Testarea unei clase necesită grijă în aranjarea contextului de test prin inițializarea corectă a clasei. Timpii de execuție pentru mai multe exemple au fost asemănători, cu metoda funcțională fiind puțin mai lentă decât cea orientată obiect pe exemple mari, dar puțin mai rapidă pe exemple mici. Un test constă în rularea funcției „has\_cycle?” pe un graf aciclic, apoi pe unul ciclic unul după altul.

|  |  |  |
| --- | --- | --- |
| Nr Noduri | Orientat Obiect (s) | Funcțional (s) |
| 100 | 0.003958 | 0.003276 |
| 10000 | 0.446022 | 0.511884 |
| 1000000 | 114.0 | 122.3 |

Lucruri care mă fac sa spun „de ce?”

Ruby se descrie ca „natural” și „gândit pentru oameni”, însă unele alegeri sintactice mi se par în opoziție totală cu aceste argumente. Comparând cu limbajul Python:

|  |  |  |
| --- | --- | --- |
|  | Ruby | Python |
| Accesul la variabilele instanței | @value | self.value |
| Apelare de metode și funcții | “hello, world!”.upcase  add = ->(x, y) do x + y end  add.call(2, 3)  add.(2, 3)  add[2, 3] | “hello, world!”.upper()  add = lambda x, y: x + y  add(2, 3) |
| Denumiri recomandate | def is\_even?()  ... | def is\_even()  ... |
| Return implicit | def greet(name)  "Hello, #{name}"  end | def greet(name):  return f"Hello, {name}" |

Aceste decizii sunt denumite de către comunitate ca „ruby magic”. În opinia mea aceste decizii îngreunează limbajul la prima vedere, chiar dacă poate fi mai eficient de scris pentru cei ce cunosc bine limbajul. Dacă aș vrea să fiu rău aș spune că Yukihiro Matsumoto a făcut anumit ca, măcar părți din limbaj, să fie asemănătoare cu limba lui maternă, japoneza.

Ruby în acțiune

Ruby este cel mai des folosit în dezvoltarea aplicațiilor web prin cel mai popular framework al limbajului, Ruby on Rails și în scripting (precum utilități de CI/CD, automatizări sau script-uri rapide) sau în managementul de configurații în cadrul multor unelte și aplicații, cum ar fi Homebrew, un manager de pachete pentru MacOS sau CocoaPods, un manager de dependențe folosit de către Xcode, IDE-ul principal pentru dezvoltarea aplicațiilor pentru sistemele de operare de la Apple. Ruby vine de asemenea instalat în avans pe sistemul de operare MacOS. În ultimii 20 de ani popularitatea este în scădere (limbajul se află în acest moment pe locul 21 în topul limbajelor populare), însă nu este considerat deloc abandonat sau mort, fiind un limbaj matur și foarte îndrăgit de comunitate.

Concluzii

Etichetarea limbajelor de programare a devenit foarte dificilă din cauza avansurilor tehnologice și creșterea popularității pentru programare în sine. Avansurile tehnologice înseamnă posibilitatea de a include mult mai multe facilități în limbaj, iar popularitatea aduce împrumutarea de facilități din alte limbaje, chiar și din unele mai puțin cunoscute sau nișate. Un limbaj de programare actual reprezintă mult mai mult decât sintaxă și compilator sau interpretor. Este o contopire complicată între comunitate, recomandările ei pentru nou veniți și stilul de abordare a problemelor, librării, framework-uri, facilitățile nu doar posibile ci și încurajate de limbaj prin ușurința cu care pot fi utilizate și mediul în care pot fi integrate soluțiile dezvoltate cu acesta.

Luând în considerare tot ce s-a descris, este evident că Ruby este un limbaj orientat obiect care nu a fost gândit cu programarea funcțională pe primul loc. Aceste facilități par să fie adăugate pe lângă, în loc să fie o parte importantă a limbajului. O întrebare care este destul de des pusă pe forumuri este daca Ruby oferă cu adevărat suport pentru *first-class functions* și dacă poate fi considerat funcțional. Internetul se află (ca de obicei) într-un continuu război pe tema asta și nici o parte nu pare să vină cu argumente finale.

Un exemplu de discuție scurtă pe tema aceasta:

A screenshot of a computer

AI-generated content may be incorrect.