LABORATORUL 7
COMPUNEREA FUNCȚIILOR

April 12, 2020

Gabriel Danciu
Universitatea Transilvania din Brasov
Departmentul de Electronica si Calculatoare

Contents

0.1 Conținutul laboratorului

1. Compunerea funcțiilor
2. Currying
3. Aplicații parțiale

0.2 Compunerea funcțiilor

Compunerea funcțiilor este un mecanism de a combina mai multe funcții simple pentru a construi alta mai complexă. Modul în care se realizează acest lucru este următorul: rezultatul fiecărei funcții este transmis următoarei. De exemplu am putea scrie *f(g(x))*. Mai întâi se evaluează *g(x)* și rezultatul este transmis lui *f*. Să presupunem că avem de caculat expresia 4 + 5 * 7. Putem crea 2 funcții pentru adunare și înmulțire:

```
const add = (a, b) => a + b;
const mul = (a, b) => a * b;
```

Evaluarea expresiei se va face ținând cont de ordinea operatorilor:

```
res = add(4, mul(5, 7))
```

Un alt exemplu simplu de compunere este cel de mai jos, unde răspundem la întrebarea: care este lista de utilizatori majori?

```
const users = [
    { name: "Ana", age: 15 },
    { name: "Ion", age: 19 },
    { name: "Dan", age: 25 },
]
const myfilter = (f, arr) => arr.filter(f);
const mymap = (f, arr) => arr.map(f);

res = mymap(u => u.name, myfilter(u => u.age >= 18, users));
```

Toate bune, însă cum putem reutiliza un alt mod de a filtra colecțiile fără a depinde de proprietățile obiectelor (exemplu age). Ne trebuie o funcție de nivel înalt, pentru că vom returna o altă funcție. Vom utiliza funcțiile ca argumente, și le vom converti într-un șir de funcții folosind operatorul...

```
const compose = (...functions) => args => functions.reduceRight((arg,fn) => fn(arg),
```

Scopul metodei "reduceRight" este de a evalua pe rând funcțiile definite ca al doilea parametru, în timp ce primul parametru reprezintă argumentele transmise funcțiilor de evaluat.

Un exemplu de utilizare a acestei metode *compose*, poate fi găsit mai jos:

```
compose(
  mymap(u => u.name),
  myfilter(u => u.age >= 18)
)(users)
```

Funcțiile *mymap* și myfilter pot fi modificate folosind currying și expresii lambda:

```
const myfilter = f => arr => arr.filter(f);
const mymap = f => arr => arr.map(f);
```

Dacă în loc de *reduceRight* am fi folosit *reduce*, atunci rezultatul apelului metodei "compose" ar fi fost altul:

```
const compose = (...functions) => args => functions.reduce((arg,fn) => fn(arg), args)
compose(
  mymap(u => u.name),
  myfilter(u => u.age >= 18)
)(users)
```

În acest caz, apelând metodele astfel, nu obținem nimic. De ce? Rezolvați astfel încât să se poată returna utilizatorii peste 18 ani. Asemănător exemplului de mai sus, scrieți o metodă pentru a calcula suma primelor n naturale pare. Încercați să creați o metodă *filter* și o metodă ce realizează *reduce*. Pe lângă aceasta, încercați să creați o funcție suma primelor numere impare. Aceste numere pot fi generate sub forma unui șir de o altă funcție.

0.3 Curring

Currying este un model de compunere a funcțiilor și înseamnă apelarea unei funcții evaluează câte un argument odată. Dacă o funcție are trei parametrii A, B și C, versiunea curry a acesteia, va prelua un argument (C), va evalua metoda și va returna rezultatul. Apoi va evalua următorul parametru (B), va returna rezultatul ș.a.m.d. Spre exemplu suma a două numere se poate rescrie astfel:

```
const add = a => b => a + b;
const res = add(2)(3);
```

Practic pentru *f(a,b) = a + b* se apelează *f(a)(b)* iar suma se returnează *b+a*.

Evident că numărul de parametrii poate varia și metoda de adunare poate fi aplicată asupra a trei sau multor parametri.

```
const curryadd = x => y => z => x + y + z
```

Un alt exemplu de currying unde parametrii sunt date (string sau int) este următorul:

```
const curriedSubstring = start => length => str => str.substr(start, length);

const getSubstring = (start, length, str)=> curriedSubstring (start) (length) (str);
const getFirstCharacters = (length, str) => curriedSubstring (0) (length) (str);

res = getSubstring(1,3,"something");
res = getFirstCharacters(4,"something");


Dar parametrii transmiși pot fi funcții:

function curry(fun) {
    return function(arg) {
        return fun(arg);
    };
}
f = curry(_.map);
res = f({1:6,2:"3",3:5});
```

În acest exemplu, metoda este transmisă abia după ce sunt evaluați parametrii, respectiv colecția 1:6,2:"3",3:5. Astfel putem privi *fun* ca pe un model de "closure" studiat în laboratoarele anterioare.

Putem rescrie metoda *compose* din secțiunea anterioară, pentru a crea compunerea a două funcții prin currying:

```
const currycompose = (...fns) =>
    fns.reduce((f, g) => (...args) => f(g(...args)));
```

Un exemplu de utilizare a acestei metode este:

```
const curriedLowerCase = str => str.toLowerCase ();
const getNewComposedFirstCharacterAsLower
    = currycompose (curriedLowerCase, curriedSubstring (0) (1));

res1 = getNewComposedFirstCharacterAsLower("DaEbsS");
res2 = curriedLowerCase("DaEbsS");
```

0.4 Aplicații parțiale

O aplicație parțială este o funcție ce a fost deja evaluată pentru unii parametrii, însă nu pentru toți. Altfel spus, unii parametri ai unei astfel de funcții sunt captați prin closure. Acei parametrii se denumesc "aplicați parțial". Și atunci care este diferența unei astfel de funcții față de una curry. Toate funcțiile curry returnează aplicații parțiale, dar nu toate aplicațiile parțiale sunt rezultatul unei funcții curry.

Să luam de exemplu metoda *add* din a doua secțiune definită astfel: *a => b => a + b;*.

Se poate crea o altă metodă ce va defini doar un singur parametru pentru *add*:

```
const increment = add(1);
```

În clipa de față metoda *increment* poate fi apelată fără niciun parametru însă rezultatul va fi NaN. Metoda *increment* va fixa parametrul *a* din cadrul metodei *add* la valoarea 1. Dar apelul de mai jos va funcționa pentru că parametrul *b* din metoda *add* va fi 3.

```
const res = increment(3);
```

Astfel metoda *add*, medotă curry are o aplicație parțială și anume *increment*.

0.5 Teme

1. Răspundeți întrebărilor de la pagina 3.

2. Generați toate numerele divizibile cu 10 de la 1 la 1000, folosind compunerea funcțiilor. Comparați timpii de execuție cu cei de la laboratorul anterior, pentru aceeași problemă.

3. Se dă o colecție de proiecte unde fiecare proiect este caracterizat de: nume, data crearii, data ultimei modificari, este activ/inactiv, detalii. Folosind funcții compuse, returnați: din toate proiectele active, informația <nume, detaliu>, ordonată crescător pe data ultimei modificări.