Racket: Funcții ca valori. Funcționale

- Responsabil: Ionuț Baciu [mailto:ionut.b096@gmail.com]

- Data publicării: 03.03.2019
- Data ultimei modificări: 19.03.2019 (+ Soluții)

Obiective

Aspectele urmărite sunt:

- Funcții ca valori de ordinul 1
- Clasificarea funcțiilor după modul de primire a parametrilor
- Reutilizare de cod
- Funcționale

Funcții ca valori de prim rang

În programarea funcțională, funcțiile sunt valori de ordinul 1. Astfel, ele pot fi manipulate ca orice altă valoare, de exemplu:

- legate la un identificator: (define par? even?)
- stocate într-o structură de date: (list < > odd? even?)
- pasate ca argumente într-un apel de funcție: (list? even?)
- returnate ca rezultat al unui apel de funcție: funcții curry din paragraful următor

Funcții curry/uncurry

O funcție care își primește toți parametrii deodată se numește funcție **uncurry**. Până acum ați folosit doar funcții uncurry.

```
 (define add-uncurry
  (lambda (x y)
    (+ x y)))
```

O funcție care returnează o nouă funcție atunci când este aplicată pe mai puține argumente decât îi sunt necesare se numește funcție **curry**.

```
(define add-curry
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

La aplicarea unei funcții pe mai puține argumente, rezultatul este o funcție care așteaptă restul argumentelor.

Funcțiile curry facilitează reutilizarea de cod, permițând obținerea unor funcții particulare din funcții mai generale:

```
(define inc-curry (add-curry 1))
```

Reutilizare de cod

În secvența de cod de mai jos sunt implementate două funcții. Prima funcție obține pătratele elementelor unei liste, iar a doua funcție obține cuburile elementelor listei.

```
(define (sq x) (* x x))
(define (cub x) (* x x x))

(define (sq-every L)
```

```
  (if (null? L)
      L
      (cons (sq (car L)) (sq-every (cdr L)))))



(define (cub-every L)
  (if (null? L)
      L
      (cons (cub (car L)) (cub-every (cdr L)))))

(sq-every '(1 2 3 4))
(cub-every '(1 2 3 4))
```

După cum se poate observa, ambele funcții folosesc același pattern:

```
(define (?nume L)
  (if (null? L)
      L
      (cons (?functie (car L)) (?nume (cdr L)))))
```

Singurele diferențe sunt numele funcției (?nume) și funcția aplicată pe fiecare element al listei (?functie).

Prin urmare, pentru a nu scrie de două ori același cod putem defini o altă funcție mai generală:

```
(define (general-func f L)
  (if (null? L)
      L
      (cons (f (car L)) (general-func f (cdr L)))))
```

Această funcție poate fi apoi folosită pentru a implementa sq-every și cub-every:

```
(define (sq-every L) (general-func sq L))
(define (cub-every L) (general-func cub L))
```

Se observă că variabila ?functie din pattern-ul detectat anterior este acum parametru al funcției general-func. Această funcție mai generală poate fi particularizată prin pasarea diverselor funcții ca parametru (precum sq sau cub).

Deoarece funcțiile sq si cub sunt folosite o singură dată, acestea pot fi scrise in-place ca funcții anonime:

```
(define (sq-every-in-place L) (general-func (lambda (x) (* x x)) L))
(define (cub-every-in-place L) (general-func (lambda (x) (* x x x)) L))
```

Dacă dorim să scriem încă o funcție care să adune 2 la fiecare element al unei liste de numere, tot ce trebuie să facem este să folosim funcția general-func:

```
(define (+2-every L) (general-func (lambda (x) (+ 2 x)) L))
```

Însă codul de mai sus mai poate fi simplificat. Nu este nevoie să definim o nouă funcție pentru adunarea cu 2 a unui număr, ci ne putem folosi de funcția de adunare deja definită, pe care o aplicăm pe un singur parametru:

```
(define (+2-every L) (general-func (add-curry 2) L))
```

Este important de observat faptul că funcția de adunare trebuie să fie curry pentru a putea fi aplicată pe un singur parametru. Astfel, putem vedea utilitatea folosirii funcțiilor curry în scopul reutilizării de cod.

Funcționale

Funcționalele (în engleză higher order functions) sunt funcții care manipulează alte funcții, primindu-le ca argumente sau returnându-le ca rezultat, după cum e nevoie. Acestea sunt foarte utile în a aplica anumite modele de calcul des folosite.

Se poate observa că funcția general-func (definită mai sus) este, de asemenea, o funcțională deoarece primește ca parametru o altă funcție. Aplicarea unei funcții pe fiecare element al unei liste reprezintă un pattern des întâlnit în programarea funcțională și este deja implementat în Racket sub numele de map:

- `map`: returnează lista rezultatelor aplicării unei funcții f asupra fiecărui element dintr-o listă. [Se pot da n liste și atunci f are n parametrii, fiecare din câte o listă. Listele trebuie să aibă aceeași lungime]

```
(map f L)
```

Mai jos se pot observa câteva exemple folosind map:

```
(map add1 '(1 4 7 10)) ; întoarce '(2 5 8 11)
(map sq '(1 2 3 4)) ; întoarce '(1 4 9 16)
(map + '(1 2 3) '(4 8 16) '(2 4 6)) ; întoarce '(7 14 25)
```

Alte exemple de funcționale des întâlnite:

- `filter`: returnează lista elementelor dintr-o listă care satisfac un predicat p.

```
(filter p L)
```

Mai jos se pot observa câteva exemple folosind filter:

```
(filter even? '(1 3 4 7 8)) ; întoarce '(4 8)
(filter positive? '(1 -2 3 4 -5)) ; întoarce '(1 3 4)
(filter (lambda (x) (>= x 5)) '(1 5 3 4 10 11)) ; întoarce '(5 10 11)
```

- `foldl` (*fold left*): returnează rezultatul aplicării funcției f pe rând asupra unui element din listă și a unui acumulator. Ordinea folosirii elementelor din listă este de la stânga la dreapta. [Se pot da n liste și atunci f are (+ n 1) parametrii, dintre care ultimul este acumulatorul. Listele trebuie să aibă același număr de elemente]

```
(foldl f init L)
```

Mai jos se pot observa câteva exemple folosind foldl:

```
(foldl cons null '(1 2 3 4)) ; întoarce '(4 3 2 1)
(foldl (lambda (x y result) (* result (+ x y))) 1 '(4 7 2) '(-6 3 -1)) ; întoarce -20
```

- `foldr` (*fold right*): singurele diferențe între foldl și foldr este că foldr ia elementele de la dreapta spre stânga și că are nevoie de un spațiu proporțional cu lungimea listei.

```
(foldr f init L)
```

Mai jos se pot observa câteva exemple folosind foldr:

```
(foldr cons null '(1 2 3 4)) ; întoarce '(1 2 3 4)
(foldr (lambda (x acc) (cons (* x 2) acc)) null '(1 2 3 4)) ; întoarce '(2 4 6 8)
```

- `apply`: returnează rezultatul aplicării unei funcții f cu argumente elementele din lista L

```
(apply f L)
```

Mai jos se pot observa câteva exemple folosind apply:

```
(apply + '(1 2 3 4)) ; întoarce 10
(apply * 1 2 '(3 4)) ; întoarce 24
(apply cons '(1 2)) ; întoarce perechea '(1 . 2)
```

Fără funcționale ar trebui să scriem mereu un cod asemănător, ceea ce nu este în spiritul reutilizării codului. Ceea ce trebuie să facem când în spatele unor prelucrări observăm un mecanism mai general, mai abstract, este să scriem o funcțională care descrie acel mecanism.

Prin folosirea funcționalelor și funcțiilor curry, codul scris este mult mai restrâns dar și mult mai clar și ușor de urmărit.

Exemple

De multe ori, funcționalele `map` și `apply` sunt încurcate. Pentru a înțelege mai bine diferența dintre acestea, urmăriți rezultatele exemplelor de mai jos.

```
(map list '(1 2 3)) ; întoarce'((1) (2) (3))
(apply list '(1 2 3)) ; întoarce '(1 2 3)
(map list '(1 2 3 4) '(5 6 7 8)) ; întoarce '((1 5) (2 6) (3 7) (4 8))
(apply list '(1 2 3 4) '(5 6 7 8)) ; întoarce '((1 2 3 4) 5 6 7 8)
```

O altă greșeală întâlnită frecvent apare atunci când funcțiile primite ca argument de către funcționale sunt plasate între paranteze.

```
(filter (odd?) '(1 2 3 4 5)) ; odd?: arity mismatch; (GREȘIT)
(filter odd? '(1 2 3 4 5)) ; întoarce '(1 3 5) (CORECT)
(foldr (+) 0 '(8 9 10)) ; GREȘIT
(foldr + 0 '(8 9 10)) ; întoarce 27 (CORECT)
(foldr ((λ (x y) (* x y))) 1 '(1 2 3 4)) ; GREȘIT
(foldr (λ (x y) (* x y)) 1 '(1 2 3 4)) ; întoarce 24 (CORECT)
```

Resurse

Citiți exercițiile **rezolvate**; apoi, rezolvați exercițiile **propuse**.

- Exerciții rezolvate și propuse
- Cheatsheet
- Soluții

Referințe

- *Structure and Interpretation of Computer Programs [https://web.mit.edu/alexmv/6.037/sicp.pdf]*, până înainte de „Using let to create local variables”
- *Mai multe functionale [http://docs.racket-lang.org/reference/pairs.html#%28part._.List_.Iteration%29]*
