LAB 5

Monday, March 28, 2022 1:14 PM

EVALUARE

Aplicativă	Leneșă
$(\lambda x.\lambda y\ (x+y)\ 1\ (\lambda z.\ (z+2)\ 3))$ $(\lambda x.\ \lambda y\ (x+1)\ 1\ 5)$ 6	$(\lambda x.\lambda y\ (x+y)\ 1\ (\lambda z.\ (z+2)\ 3))$ $1 + (\lambda z.\ (z+2)\ 3)$ $1 + 5$ 6
-> se evaluează întâi parametrii funcției -> "vreau rezultatul!" (eager evaluation)	-> întâi se aplică funcția -> se întârzie evaluarea param. până când chiar este nevoie de rezultat -> "mai aștept" (lazy evaluation)
Racket	Haskell
transferul param: prin valoare	transferul param: prin nume

Exemplu

$Fix = \lambda f.\ (f\ (Fix\ f))$

$ct = \lambda x.\ b$

Aplicativă	Leneșă
$(Fix\ ct) = (\lambda f.\ (f\ (Fix\ ct))\ ct)$ $= ct\ (Fix\ ct)$ => ev. parametrul $= ct\ (\lambda f.\ (f\ (Fix\ ct)))\ ct)$ $= ct\ (ct\ (Fix\ ct))$ $= ct\ (ct\ (\lambda f\ (f\ (Fix\ ct))\ ct)$ $= ct\ (ct\ (ct\ (Fix\ ct)))$	$(Fix\ ct) = \lambda f.\ (f\ (Fix\ ct)\ ct)$ $= ct\ (Fix\ ct)$ $= \lambda x.b\ (Fix\ ct)$ $= b$ • param. nu mai e evaluat, pt că nu mai e nevoie de el

Cum putem "traduce" evaluarea leneșă în Racket?

ÎNCHIDERI	PROMISIUNI
(define sum (lambda(x y) (lambda () (+ x y)))) (sum 1 2) => Ce întoarce acest apel ? => o funcție Pentru a **forța** evaluarea ((sum 1 2))	(define sum (lambda(x y) (delay (+ x y)))) (sum 1 2); *va afișa #<promise>* Pentru a **forța** evaluarea (force (sum 1 2))

FLUXURI

= obiecte infinite / șiruri

-> se bazează pe evaluare leneșă

= (elem. curent , generator)

primul elem. — promisiune / închidere fct.

```
(define ones-stream
 (cons 1 (lambda () ones-stream)))
```

1 — elem. curent; (lambda () ones-stream) — generator (închidere fct)

```
(define ones-stream
 (cons 1 (delay ones-stream)))
```

1 — elem. curent; (delay ones-stream) — generator (promisiune)

Ce se întâmplă?

-> generatorul produce valorile atunci când este nevoie de ele

Liste	Fluxuri
cons	stream-cons
car	stream-first
cdr	stream-rest
'()	empty-stream
null?	stream-empty?
map / filter	stream-map / stream-filter

-
```
(define (stream-take s n)
    (cond ((zero?  n) '())
          ((stream-empty? s)' ())
          (else (cons (stream-first s)
                      (stream-take (stream-rest s) (- n 1))))))
```

-
```
(define (make-naturals k)
    (stream-cons  k  (make-naturals (add1 k))))
```

el. curent — generator

```
(define naturals-stream (make-naturals 0))

(stream-take  naturals-stream  4)
```

stream-take (0, gen0) 3 = (cons (stream-first (0, gen0))
(streak-take (stream-rest (0, gen0)) 2)))

→ Ce întoarce (stream-rest (0, gen0)) => gen0
→ Ce este gen0? => un stream
→ gen0 = (make-naturals (add1 0))
= (make-naturals 1)
= stream-cons 1 (make-nat (add1 1))
gen 1
= (1, gen1) = un nou stream

stream-map → cum putem genera fluxuri

$$\begin{cases} s = x_0, x_1, x_2, \dots \\ map(f, s) = f(x_0), f(x_1), f(x_2) \dots \end{cases}$$

$s = 1, map(f, s) \qquad f(x) = x + 2$

$\Rightarrow s = 1, f(x_0), f(x_1), f(x_2) \dots \qquad x_0 = 1$

$s = 1, f(1), f(x_1), \dots$

$s = 1, 2, f(x_1), \dots \qquad x_1 = 2$

$s = 1, 2, f(2), \dots$

$s = 1, 2, 4, \dots$

stream-take (0, gen0) 3 = (cons (stream-first (0, gen0))
(stream-take (stream-rest (0, gen0)) 2)))
= (cons 0 (stream-take (1, gen1) 2))
= (cons 0 (cons 1 (stream-take (2, gen2) 1)))
= (cons 0 (cons 1 (cons 2 (stream-take (3, gen3) 0))))
= (cons 0 (cons 1 (cons 2 '())))
= '(0 1 2)

Exerciții

Obs: Când generăm stream-uri ne gândim cum am face pt. liste / cum putem folosi alte fluxuri pe care să le prelucrăm

(ex 2)
```
stream-zip-with f () s2 = ()
stream-zip-with f s1 () = ()
stream-zip-with f h1:t1 h2:t2 = (f h1 h2):
                                (stream-zip-with f t1 t2)
```

h_1 => stream-first s_1
t_1 => stream-rest s_2
: => stream-cons

ex 4 $1 \quad -\frac{1}{3} \quad \frac{1}{5} \quad -\frac{1}{7} \quad \dots$

① explicit => { n: 1 3 5 7 ; sign: 1 -1 1 -1 } ; la fiecare pas: n+2, -sign

hint: named let with 2 params (n, sign)

② implicit

=> putem plica de la { stream-ul nr. impare => stream-filter ; stream-ul 1 -1 1 -1 ... => repeat }

hint: stream-zip-with

ex 5 $x, (f_1\ x), (f_2\ y), (f_1\ z) \dots$ (y, z)

stream-zip-with

$s_1 = x_0, x_1, x_2 \dots = f_1, f_2, f_1, f_2 \dots$

$s_2 = y_0, y_1, y_2 \dots = y_0, \text{map}(?, s_2)$

=> nu putem folosi map

$s_2 = x_0, (\text{stream-zip-with}\ f\ s_1\ s_2^*)$

apply: $f_1, f_2, \dots$ aplicate pe $x_0, x_1, \dots$

apply f_1 pe $[x_0]$, f_2 pe $[x_1]$, f_1 pe $[x_2]$, f_2 pe $[x_3]$

=> $s_2^* = \text{stream-map}\ \text{list}\ s_2$

ex 6 => vrem să folosim 'play' de la ex 5.

play x f_1 f_2

m

↳ λm -> if m%.4 == 0
then m-2
else m-1

↳ λm -> if m%.3 == 0
then m-1
else m-3

=> f_1 și f_2 spun ce face jucătorului 1/2 la fiecare pas (în funcție de m)

=> rezultatul este un stream de elemente (cât este m după fiecare rundă)

s_1 : m, m_1, m_2, m_3 ....

"cât este în după
runda 1 (când a jucat jucătorul 1)"

s_2 : 2, 1, 2, 1, 2 ....

=> Ne dorim un stream de perechi (m_i, jucător_curent)

$(m,2), (m_1,1), (m_2,2), (m_3,1)$ ...

m_1 curent — jucătorul 1

(stream-zip-with cons (s_1 : m, m_1, m_2, m_3 ; s_2 : 2, 1, 2, 1 ...)) = s_3

Ne dorim ca din s_3 să extragem numai perechile în care $m_i \leq 0$
=> stream-filter => s_4

Ne dorim ca din s_4 să extragem numai prima pereche $(m_i, jucator)$, iar din această pereche, îl extragem pe jucător (care este 1 sau 2)

s_1 : $m, m_1, m_2, \ldots (m_i \ldots$
s_2 : 2, 1, 2, $\ldots (j_i \ldots$)

$m_i \leq 0$
(m_i, j_i) => j_i

ex 7-10 freestyle

ex 7 - 10 freestyle