LAB2 - notes

Wednesday, March 9, 2022 12:27 PM

RECURSIVITATE

① PE STIVĂ

fact 1 = 1
fact n = n * fact(n-1)

=> rezultatul apelului recursiv este folosit într-o expresie mai complexă

fact 3 = 3 * fact(2)
= 3 * (2 * fact(1))
= 3 * (2 * (1 * fact(0)))
= 3 * (2 * (1 * 1))

← mă întorcem din rec

= 3 * (2 * 1)
= 3 * 2
= 6

→ rezultatul final se calculează la întoarcerea din recursivitate

② PE COADĂ

→ "tail" call => apelul recursiv este chiar rezultatul (nu îl mai folosim în altă expresie)
→ nu mai avem nevoie de informații pe stivă
→ construim rezultatul odată cu avansul în recursivitate
→ păstrăm rezultatul într-un parametru (acc)

fact-helper 1 acc = acc → când ieșim din rec., rezultatul este chiar ce am "strâns" în acc
fact-helper n acc = fact-helper n-1 acc*n

fact-tail n = fact-helper n 1

fact-helper 3 1 = fact-helper 2 (3*1)
= fact-helper 1 2*3
= 2*3

③ ARBORESCENTĂ → pe stivă / coadă

=> cel puțin 2 apeluri recursive care se execută independent

5T

fibo 0 = 0
fibo 1 = 1
fibo n = (fibo n-1) + (fibo n-2)

CA
```
fib-helper 0 a b = a
fib-helper n a b = fib-helper n-1 b a+b
```

EXERCIȚII LAB

① REVERSE

ST

```
rev [] = []
rev (h:H) = rev (H) ++ [x]
```

(append)

CA

```
rev-helper ([] , acc) = acc
rev-helper (h:H , acc) = rev-helper (H , h: acc)
```

```
h,H :  1 [2,3]      (2),[3]      (3),[]
l   :  [1,2,3] -> [2,3] -> [3] -> []
rez :  [3,2,1] <- [3,2] <- [3] <- []
```

```
h,H    1 [2,3]      2 [3]      3 []      []
L:     [1,2,3] -> [2,3] -> [3] -> []
acc:   [1]     -> [2,1]   -> [3,2,1] -> [3,2,1]
```

② LESSER -> STIVĂ

-> stivă => rezultatul ap. recursiv este folosit într-o expr.

[1, 7, 5, 8] -> h = 6

[1,7,5,8] -> [7,5,8] -> [5,8] -> [8] -> []

[1] <- [5] <- [5] <- [] <-

③ greater -> COADĂ

=> rezultatul ap. recursiv este chiar rezultatul ap. curent => tail call

[1,7,5,8] h = 6

[1,7,5,8] -> [7,5,8] -> [5,8] -> [8] -> []

① [] -> [7] -> [7] -> [7,8] -> [7,8]

② [] -> [7] -> [7] -> [8,7] => [7,8]

rev

Pt. a construi rezultatul din acc, putem alege 2 variante:

① (ineficient) -> cu append
② (eficient) -> cu cons

⇓

- rez. va fi inversat => când gj. la cazul de bază, trb. să inversăm pe acc

(4) RM. DUPLICATES LEFT

[1,2,3,2,4,4] => [1,2,3,4]

L: [1,2,3,2,4,4] -> [2,3,2,4,4] -> [3,2,4,4] -> [2,4,4] -> [4,4] -> [4] -> []

[1,3,2,4] <- [3,2,4] <- [3,2,4] <- [2,4] <- [4] <- [4] <-

=> nu a mers pe care îl aveam => inversăm pe coadă

acc: [1] -> [1,2] -> [1,2,3] -> [1,2,3] -> [1,2,3,4] -> .... ✓

=> pe coadă => verific dacă elem. curent apare în acc

=> folosim fct. member

• și aici dacă folosim cons, trb. să inversăm rezultatul la sf.

(5) RM DUPLICATES RIGHT

=> pe stivă (ca în ex. de mai sus)

(6) SIERPINSKI

Δ => triangle L "solid" color

ΔΔ => beside

Δ
ΔΔ => above

```
(define SMALLEST-SIZE 10)
(define SMALLEST-TRIANGLE (triangle SMALLEST-SIZE "outline" "blue"))

(define (triangles-1 size)
  (if (<= size SMALLEST-SIZE)
      SMALLEST-TRIANGLE
      (above (triangles-1 (/ size 2))
             (beside (triangles-1 (/ size 2))
                     (triangles-1 (/ size 2))))))

(define (triangles-2-helper size img)
  (if (<= size SMALLEST-SIZE)
      img
      (triangles-2-helper (/ size 2)
                          (above img (beside img img)))))

(define (triangles-2 size)
  (triangles-2-helper size SMALLEST-TRIANGLE))
```

} STIVĂ

} COADĂ

[1, 2, 3, 4, 5] n = nr. iterații

(car colors) → culoare

si 0 L colors = triangle L → CB: pt. 0 iterații => desenăm 1 triunghi cu prima culoare

si n L colors = (above tr_sus (lasi de tr_stg tr_dr))

tr_sus => si n-1 L|2 colors [1,2,3,4,5] (1)

tr_stg => si n-1 L|2 .... [1,2,3,4,5] => [2,3,4,5,1] (2)

tr_dr => si n-1 L|2 .... [1,2,3,4,5] => [3,4,5,1,2] (3)

(1) => colors

(2) => (cdr colors) ++ [(car colors)]

(3) => (cdr (cdr colors)) ++ [(car colors)] ++ [(car (cdr colors))]

[3,4,5] [1] [2] [2,3,4,5]

APPEND [1,2,3] [4,5] => [1,2,3,4,5]

ST [1,2,3] [4,5] → rămâne așa, nu o modific

[1,2,3] → [2,3] → [3] → []

[1,2,3,4,5] ← [2,3,4,5] ← [3,4,5] ← [4,5]

append [] L = L

append h:tl L = h: (append tl L)

CA A: [1,2,3] B: [4,5]

B: [4,5] → [5] → []

[1,2,3] → [4,1,2,3] → [5,4,1,2,3] → reverse => [3,2,1,4,5]

=> A trb să plece inversat => [3,2,1]

app-helper A [] = reverse A

app-helper A h:t = app-helper h:A t

app-tail A B = app-helper (reverse A) B

REMEMBER

① Cum determinăm dacă recursivitatea este pe stivă / coadă?

- mă uit unde apare **apelul recursiv**
  - dacă apare singur (ex) $f\ p_1 \dots p_n = f\ p_1' \dots p_n'$ ⇒ COADĂ
  - dacă apare într-o expresie (ex) $f\ p_1 \dots p_n = (f\ p_1' \dots p_n') \pm \text{ceva}$ / $\pm \text{ceva}$ $= g\ (f\ p_1' \dots p_n')$ ⇒ STIVĂ
    - g → altă funcție
- $p_1 \dots p_n$ = parametrii funcției
- $p_1' \dots p_n'$ = parametrii noi ai funcției (din ap. recursiv)

② Faptul că o funcție are acumulator, NU înseamnă că este recursivă pe coadă.

(ex)
```
f 0 acc = acc
f n acc = 1 + (f n-1 acc+1)
```

Aici apelul recursiv apare într-o expr. mai complexă ⇒ rec. pe STIVĂ

- Cum decurge evaluarea pt (f 3 0)?

m: 3 → 2 → 1 → 0

acc 0: 1 → 2 → 3 → 3

6 ←(+1) 5 ←(+1) 4 ←(+1)

↳ rezultatul este obținut la întoarcerea din recursivitate

③ Atunci când operăm pe liste, iar la fiecare pas trebuie să adăugăm elemente în acc:

① → append ⇒ mai ineficient

```
acc ++ [h]
```

② → cons ⇒ mai eficient

```
h : acc
```

⇒ rezultatul este inversat

⇒ pe cazul de bază inversăm acc ⇒ (reverse acc)

(Vezi explicația din curs) ⇒ reverse vs append

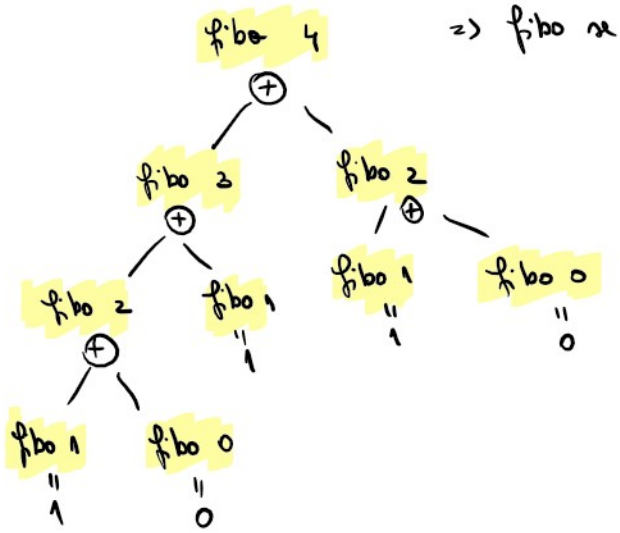
④ Recursivitate ARBORESCENTĂ

⇒ dacă ∃ cel puțin 2 apeluri rec. care se execută independent

ex: fibo 0 = 0

fibo 1 = 1

fibo n = (fibo n-1) + (fibo n-2)



⇒ fibo se apelează de 9 ori

⑤ COMPARAȚIE

- Recursivitate pe stivă: de obicei, . . .
  - Elegantă
  - Ineficientă spațial și/ sau temporal ⇒ poate cauza stack overflow
- Recursivitate pe coadă: de obicei, . . .
  - Mai puțin lizibilă decât cea pe stivă
  - Necesită prelucrări suplimentare (e.g. inversare)
  - Eficientă spațial și/ sau temporal