

LAB 2 : RECURSIVITATE

Recursivitate : 

The diagram illustrates three forms of recursion:

- mu recursive**: Represented by a bracket under the label "mu" and the word "stivă".
- mu iterative**: Represented by a bracket under the label "mu" and the word "bada".
- arboriscent**: Represented by a bracket under the label "arboriscent".

① Rec. per stiva'

- păstrează informații și stocă pe parcursul avansului în nausinicitate
 - construiește rezultatul la revenirea din nausinicitate

E
x

```
(define (list-sum-stack L)
  (if (null? L)
      0
      (+ (car L) (list-sum-stack (cdr L))))))
```

The diagram illustrates the recursive evaluation of the list-sum expression '(1 2 3 4)'. It shows four stages of evaluation:

- Stage 1:** The expression is '(1 2 3 4)'. A red arrow points from the right towards the left, indicating the direction of evaluation.
- Stage 2:** The expression is split into '(1 2)' and '(3 4)'. The result '(1 2)' is underlined in red as '2'.
- Stage 3:** The expression is split into '(1)' and '(2 3)'. The result '(2 3)' is underlined in red as '3'.
- Stage 4:** The expression is split into '(1)' and '(2)'. The result '(2)' is underlined in red as '2'.

! Complexitate [temporală: $O(k)$
 $k = \text{length } L$] - spatială: $O(a)$

sp. pe shba'

=> implemente spatial

=> posibile genera stack-overflow

Q Rec. și codă

→ Rec. se bazează pe parcursul avansului în recursitate

TAIL-CALL OPTIMIZATION = apel

recursiv direct și folosește o
ayuden următoare.

→ acc → param în care calc.
rezultatul (în general)
[acc 0]

Ex:

```
(define (list-sum-tail-helper L acc)
  (if (null? L) acc
      list-sum-tail-helper (cdr L) (+ acc (car L))))
```

```
(define (list-sum-tail L)
  (list-sum-tail-helper L 0))
```

} wrapper

list-sum-tail '(2 3 5) 0

[

list-sum-tail '(3 5) 2

|

list-sum-tail '(4) 5

|

list-sum-tail '() 9

= 9

! Complessità [temporale: $O(n)$
 $n = \text{length } L$]
spaziale: $O(1)$

\Rightarrow d.p. d. e spazial, se gli
loads e mai efficienti

Ex:

my-member $\in L - \# \neq$
 $\neq f.$

my-member $\in () = \# f$

my-member $\in [_ : L] = \neq f$

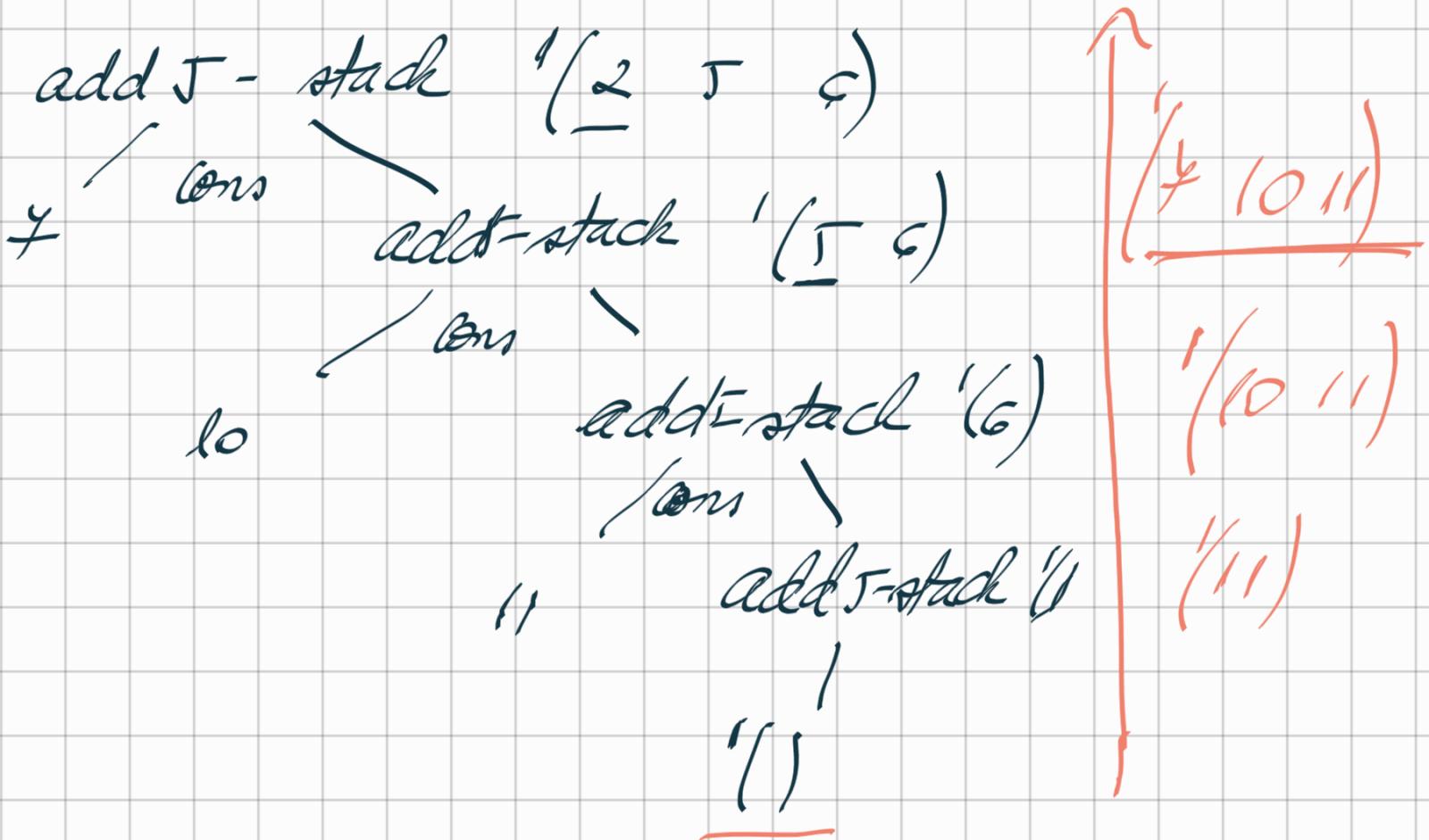
my-member $\in [y : L] =$
mymember $\in L$

! acc. poche false

SPVAT (w) COADAT

① SPVAT

```
(define (add5-stack L)
  (if (null? L)
      '()
      (cons (+ 5 (car L)) (add5-stack (cdr L))))))
```



Q COAD

```
(define (add5-tail L [acc '()])
  (if (null? L)
      acc
      (add5-tail (cdr L) (cons (+ 5 (car L)) acc))))
```

append acc (list (+ 5 (car L)))

$\text{add5-tail } \underline{(\underline{2} \ \underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{6})}$	$\text{add-tail } \underline{()}$
$\text{add5-tail } \underline{(\underline{2} \ \underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{6})}$	$\text{add-tail } \underline{()}$
$\text{add5-tail } \underline{(\underline{2} \ \underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{6})}$	$\text{add-tail } \underline{()}$
$\text{add5-tail } \underline{(\underline{2} \ \underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{5} \ \underline{6})}$	$\text{add-tail } \underline{(\underline{6})}$	$\text{add-tail } \underline{()}$

// // 10 2

① reverse acc

② append

① reverse acc

$$O(u) + \underbrace{O(n)}_{\text{reverse}} \Rightarrow O(u)$$

② append

A

B



$O(\text{length}(A))$

$$O(u) + O(1) + 2 + \dots + (u-1) \\ \Rightarrow O(u^2)$$

③ Rec. arborescens

→ al putin 2 apărere recursive independentă

$$\text{Fibonacci } 0 = 0$$

$$\text{Fibonacci } 1 = 1$$

$$\text{Fibonacci } n = \underline{\text{Fibonacci}(n-1)} + \underline{\text{Fibonacci}(n-2)}$$

$$\text{fibo}(4) \quad 3$$

