PARADIGME DE PROGRAMARE

Curs 2

Recursivitate pe stivă / pe coadă / arborescentă. Calcul Lambda.

1

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

2

2

Recursivitate

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive).

Recursivitate

- Singura modalitate de a prelucra date de dimensiune variabilă (în lipsa iterației)
- Elegantă (derivă direct din specificația formală / din axiome)
- Minimală (cod scurt, ușor de citit)
- Ușor de analizat formal (ex: demonstrații prin inducție structurală)
- Poate fi ineficientă: Se așteaptă rezultatul fiecărui apel recursiv pentru a fi prelucrat în contextul apelului părinte. Astfel, contextul fiecărui apel părinte trebuie salvat pe stivă pentru momentul ulterior în care poate fi folosit în calcul.

Problema

Pentru o lizibilitate sporită și aceeași putere de calcul (v. Teza lui Church), plătim uneori un preț mai mare (consum mare de memorie care poate duce chiar la nefuncționare – stack overflow).

3

3

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

4

4

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

rezultatul apelului recursiv este așteptat
← pentru a fi înmulțit cu n

```
>(fact-stack 3)
```

← apelul curent este marcat cu verde
ceea ce tocmai s-a depus pe stivă e marcat cu roz
ceea ce urmează să se scoată de pe stivă e marcat cu mov

Stiva procesului

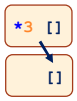
5

5

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
```



Stiva procesului

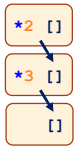
6

6

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
```



Stiva procesului

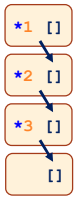
7

7

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
>(* 3 (* 2 (* 1 (fact-stack 0))))
```



Stiva procesului

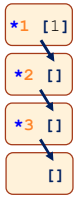
8

8

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
>(* 3 (* 2 (* 1 (fact-stack 0))))
>(* 3 (* 2 (* 1 1)))
```



Stiva procesului

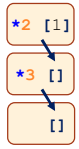
9

9

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
>(* 3 (* 2 (* 1 (fact-stack 0))))
>(* 3 (* 2 (* 1 1)))
>(* 3 (* 2 1))
```



Stiva procesului

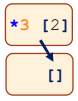
10

10

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
>(* 3 (* 2 (* 1 (fact-stack 0))))
>(* 3 (* 2 (* 1 1)))
>(* 3 (* 2 1))
>(* 3 2)
```



Stiva procesului

11

11

Exemplu – recursivitate pe stivă

```
1.  (define (fact-stack n)
2.    (if (zero? n)
3.        1
4.        (* n (fact-stack (- n 1)))))
```

```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
>(* 3 (* 2 (* 1 (fact-stack 0))))
>(* 3 (* 2 (* 1 1)))
>(* 3 (* 2 1))
>(* 3 2)
>6
```

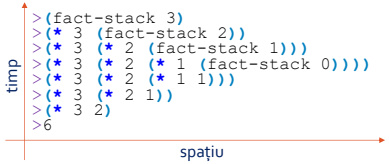
Stiva procesului

12

12

Observații – recursivitate pe stivă

- **Timp:** Θ(n) (se efectuează n înmulțiri și stiva se redimensionează de 2*n ori)
- **Spațiu:** Θ(n) (ocupat de stivă)
- **Calcul:** realizat integral la revenirea din recursivitate
- **Stiva:** reține contextul fiecărui apel părinte, pentru momentul revenirii (starea programului se regăsește în principal în starea stivei)



13

13

Comparație cu rezolvarea imperativă

```
1.  int i,  factorial = 1;
2.  for (i = 2; i <= n; i++)
3.      factorial *= i;
```

- **Timp:** Θ(n) (se efectuează n înmulțiri)
- **Spațiu:** Θ(1) (în orice moment, în memorie sunt reținute doar 3 valori, pentru variabilele `i`, `n`, `factorial`)
  - Rezultatul se construiește în variabila `factorial` pe măsură ce avansăm în iterație
  - Putem obține același comportament într-o variantă recursivă?

14

14

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

15

15

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

rezultatul se construiește în variabila `fact` pe măsură ce avansăm în recursivitate

rezultatul apelului recursiv este ← rezultatul final al funcției

```
>(fact-tail-helper 3 1)
```

Stiva procesului

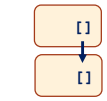
16

16

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
```



Stiva procesului

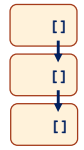
17

17

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
```



Stiva procesului

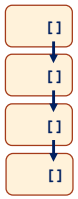
18

18

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
>(fact-tail-helper 0 6)
```



Stiva procesului

19

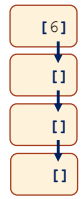
19

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
>(fact-tail-helper 0 6)
>6
```

rezultatul celui mai adânc apel recursiv se va transmite neschimbat către apelul inițial



Stiva procesului

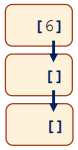
20

20

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
>(fact-tail-helper 0 6)
>6
```



Stiva procesului

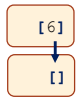
21

21

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
>(fact-tail-helper 0 6)
>6
```



Stiva procesului

22

22

Exemplu – recursivitate pe coadă

```
1.  (define (fact-tail n)
2.    (fact-tail-helper n 1))
3.
4.  (define (fact-tail-helper n fact)
5.    (if (zero? n)
6.        fact
7.        (fact-tail-helper (- n 1) (* n fact))))
```

```
>(fact-tail-helper 3 1)
>(fact-tail-helper 2 3)
>(fact-tail-helper 1 6)
>(fact-tail-helper 0 6)
>6
```

Stiva procesului

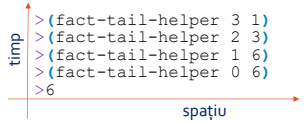
23

23

Observații – recursivitate pe coadă

Creșterea stivei nu mai este necesară. Rezultatul unui nou apel recursiv nu mai este așteptat de apelul părinte pentru a participa la un nou calcul, ci este chiar rezultatul final. Astfel, contextul apelului părinte poate fi șters din memorie. Această optimizare se numește **tail-call optimization** și este realizată de un compilator inteligent care detectează situația în care apelul recursiv este „la coadă" (nu mai participă la calcule ulterioare).

- **Timp:** $\Theta(n)$ (se efectuează n înmulțiri)
- **Spațiu:** $\Theta(1)$ (ocupat de variabilele `n` și `fact`, care reține starea programului)
- **Calcul:** realizat integral pe avansul în recursivitate



rezultatul tuturor celor 4 apeluri este
← rezultatul final al funcției, anume 6

24

24

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

25

25

Exemplu – recursivitate arborescentă

```
1.  (define (fibo-stack n)
2.    (if (< n 2)
3.        n
4.        (+ (fibo-stack (- n 1)) (fibo-stack (- n 2)))))
```

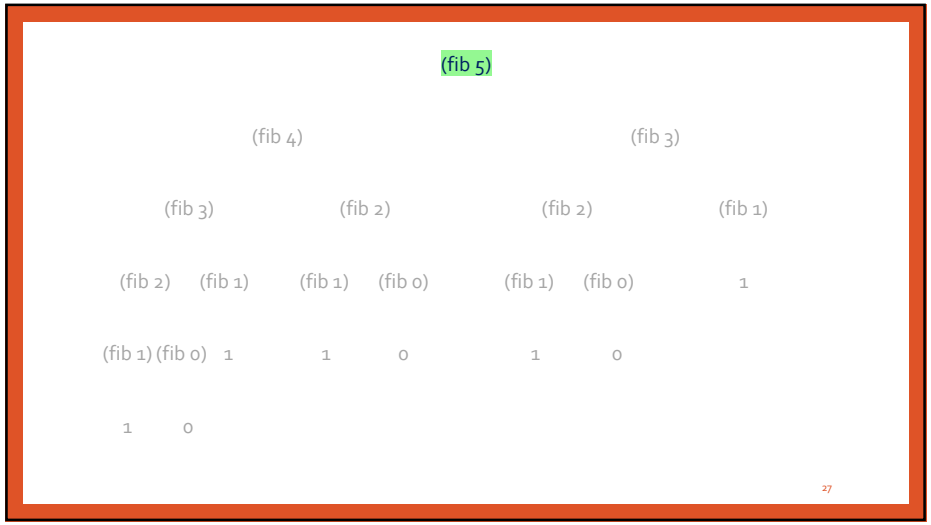
rezultatele a 2 apeluri recursive sunt așteptate pentru a fi adunate între ele

```
>(fibo-stack 3)
> (fibo-stack 2)
> >(fibo-stack 1)
< <1
> >(fibo-stack 0)
< <0
< 1
> (fibo-stack 1)
< 1
<2
```

←`(fibo-stack 1)` se calculează de 2 ori, iar numărul de calcule redundante crește exponențial cu n

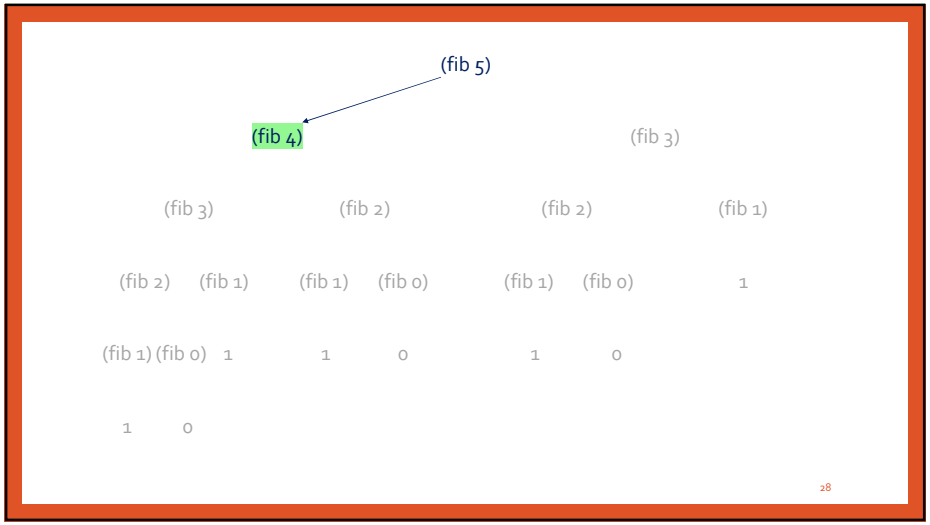
26

26



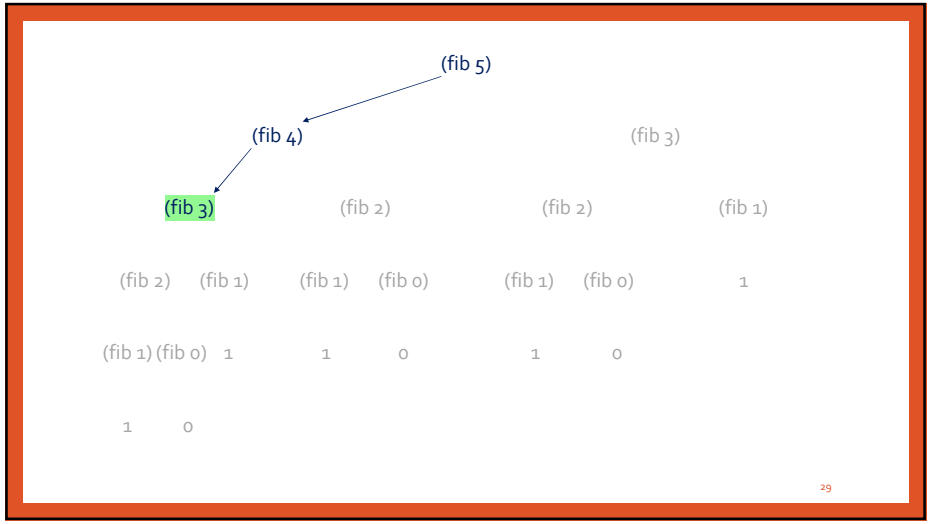
27

27

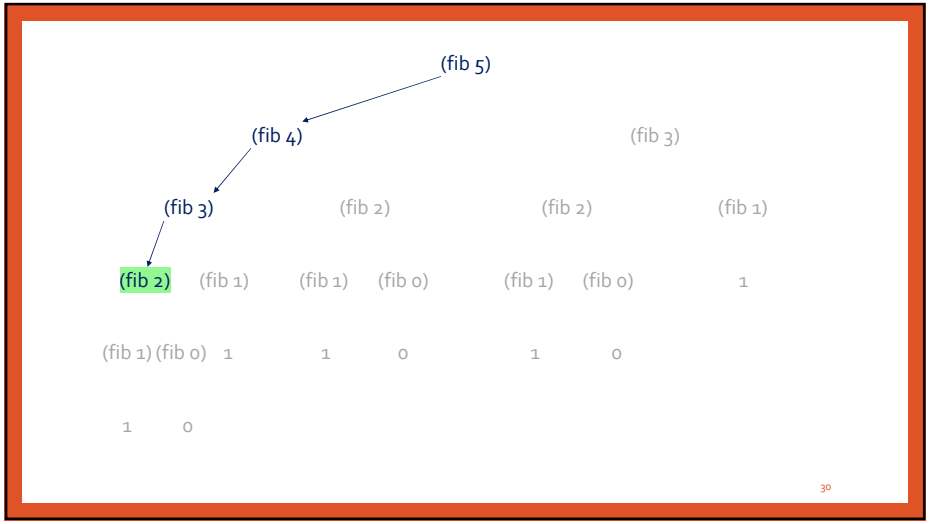


28

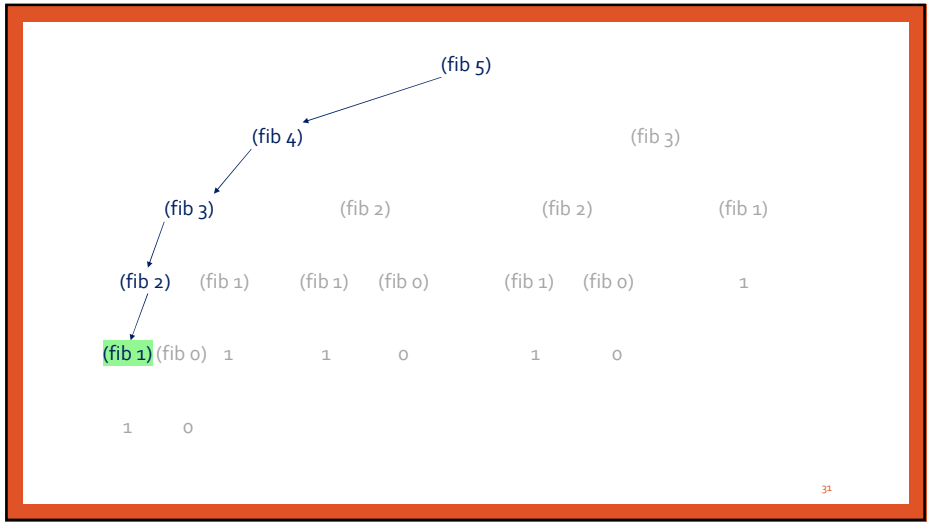
28


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2) (fib 1) (fib 1) (fib 0) (fib 1) (fib 0) 1
(fib 1) (fib 0) 1 1 0 1 0
1 0
29

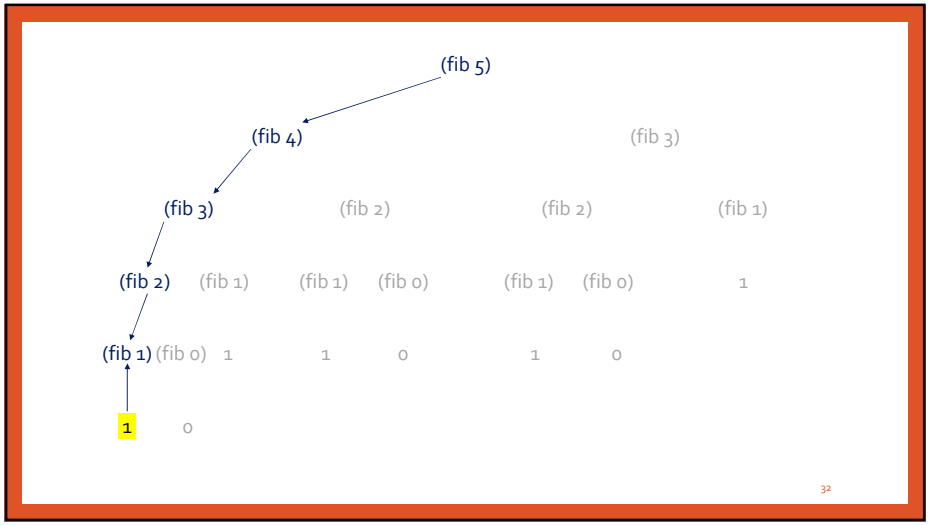
29


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2) (fib 1) (fib 1) (fib 0) (fib 1) (fib 0) 1
(fib 1) (fib 0) 1 1 0 1 0
1 0
30

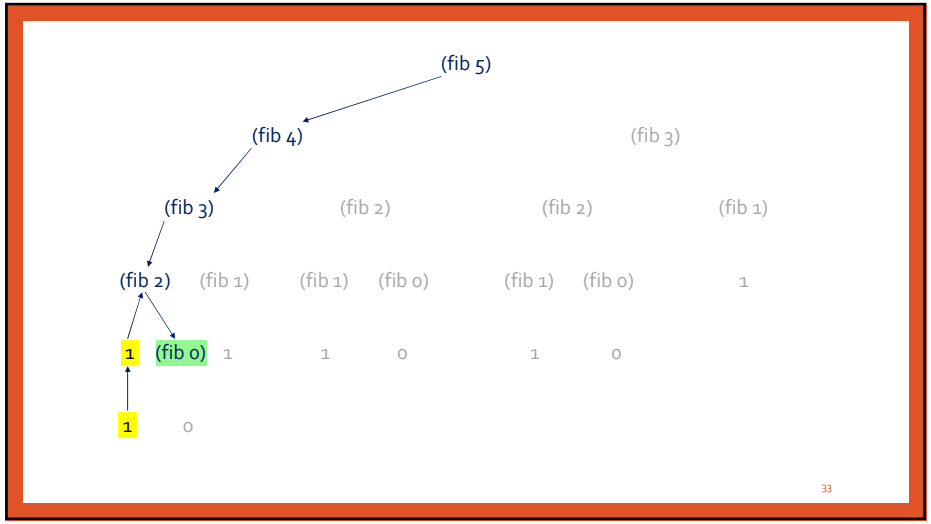
30


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2) (fib 1) (fib 1) (fib 0) (fib 1) (fib 0) 1
(fib 1) (fib 0) 1 1 0 1 0
1 0
31

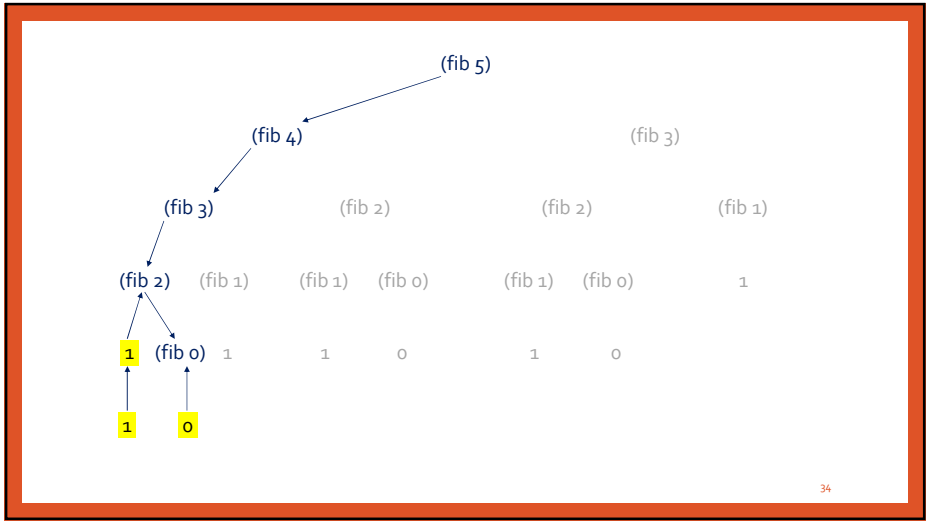
31


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2) (fib 1) (fib 1) (fib 0) (fib 1) (fib 0) 1
(fib 1) (fib 0) 1 1 0 1 0
1 0
32

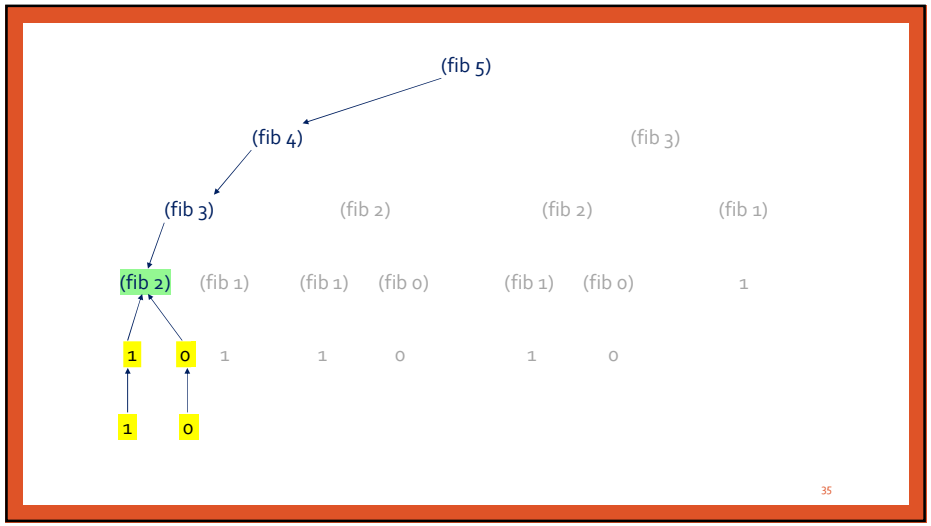
32


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
(fib 0)
1
1
0
1
0
1
0
33

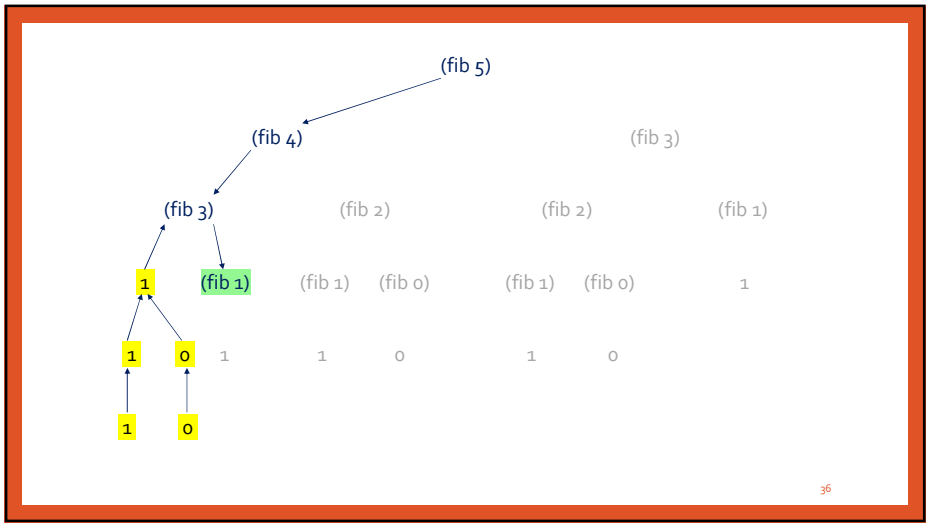
33


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
(fib 0)
1
1
0
1
0
1
0
34

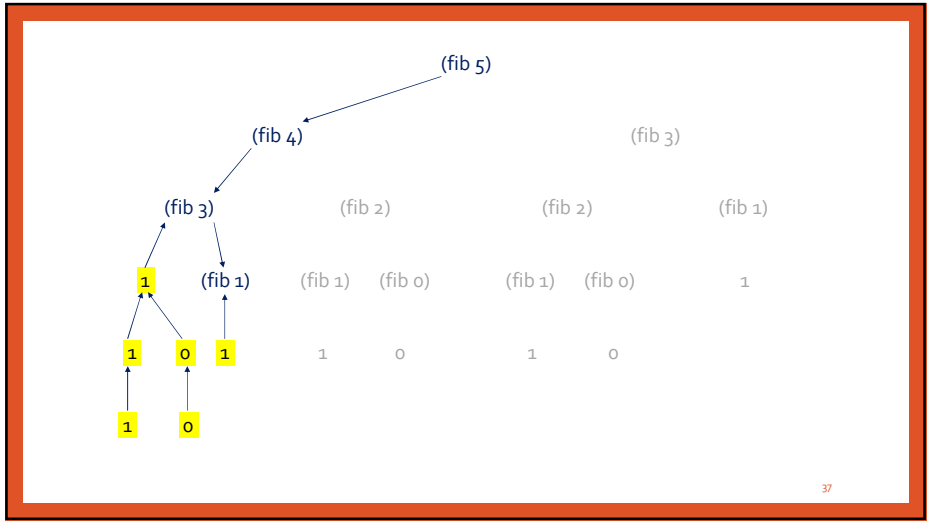
34


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0
35

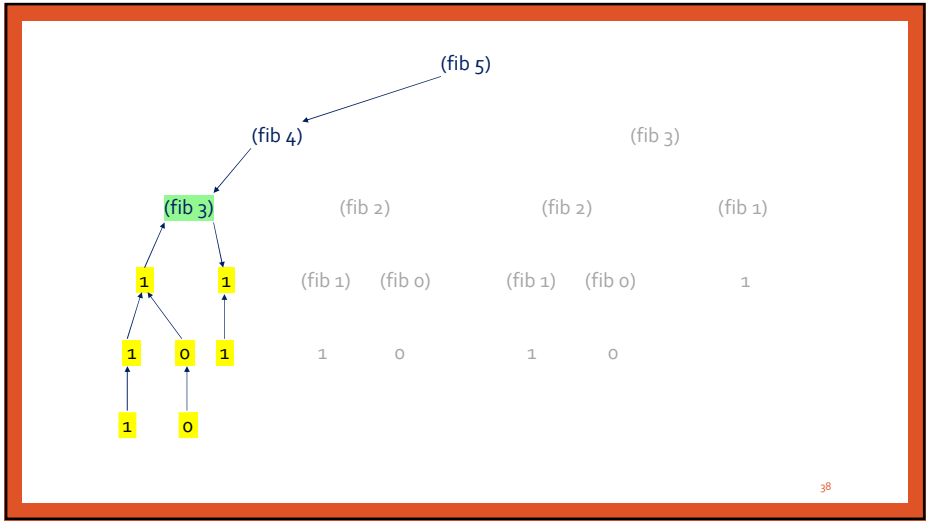
35


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0
36

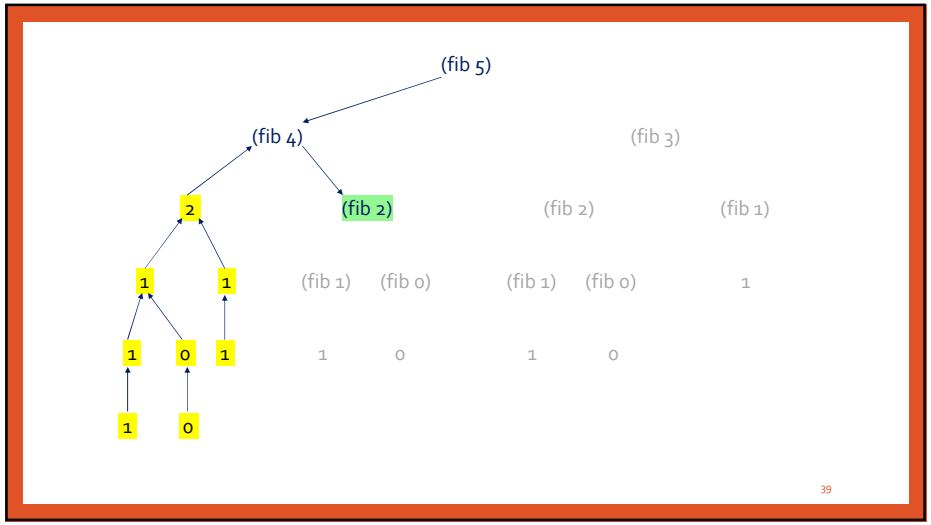
36


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
37

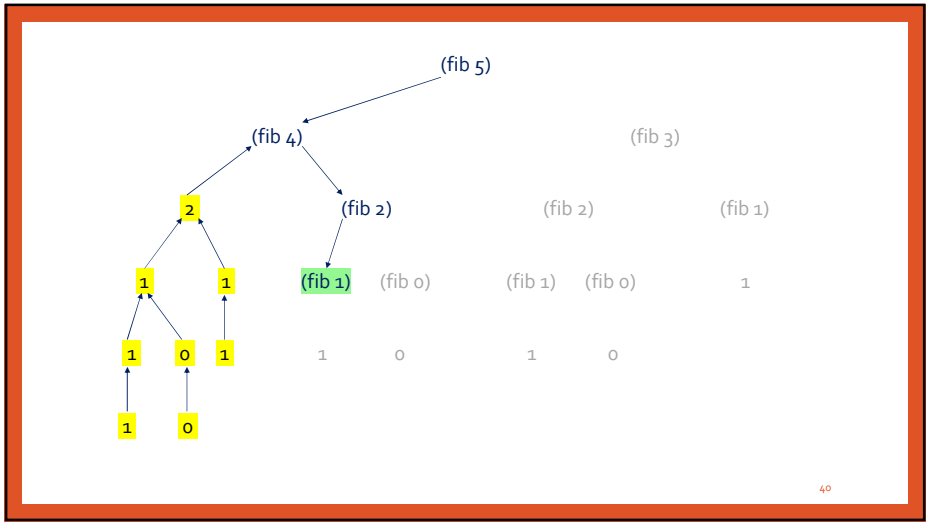
37


(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
38

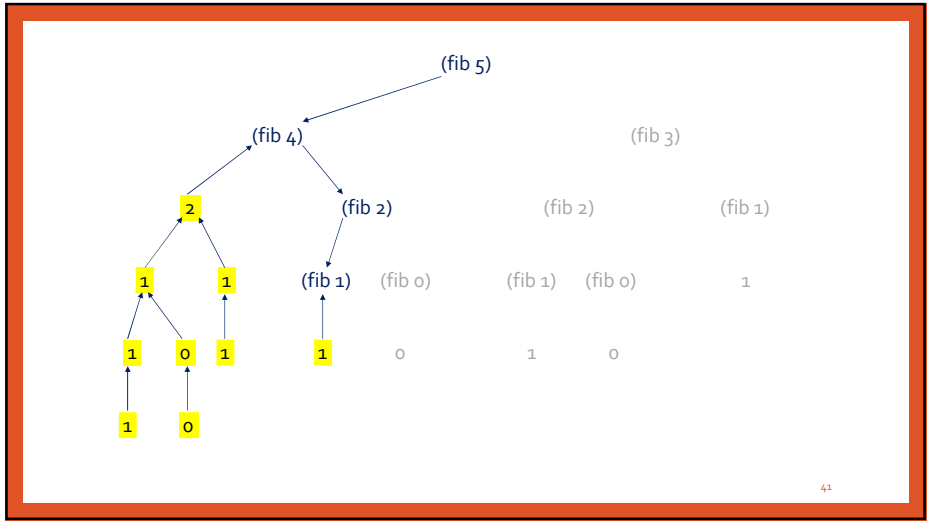
38


(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
39

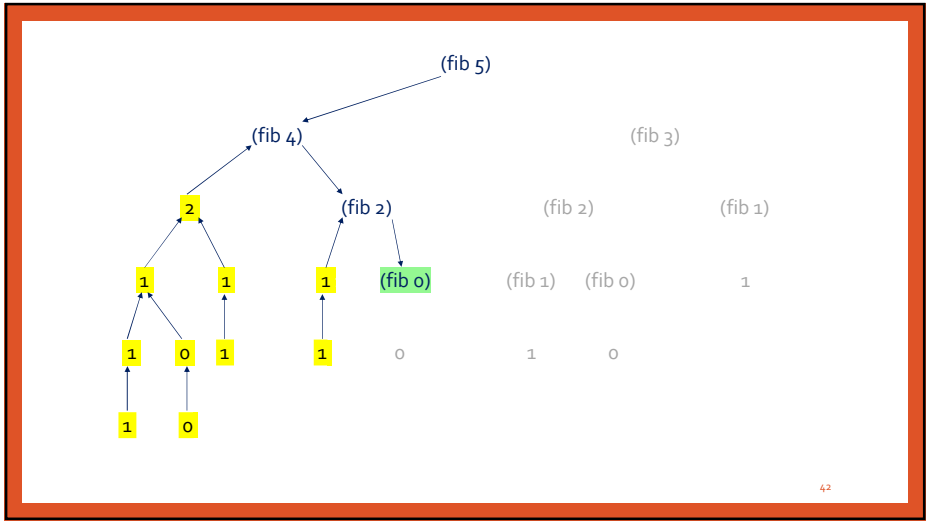
39


(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
40

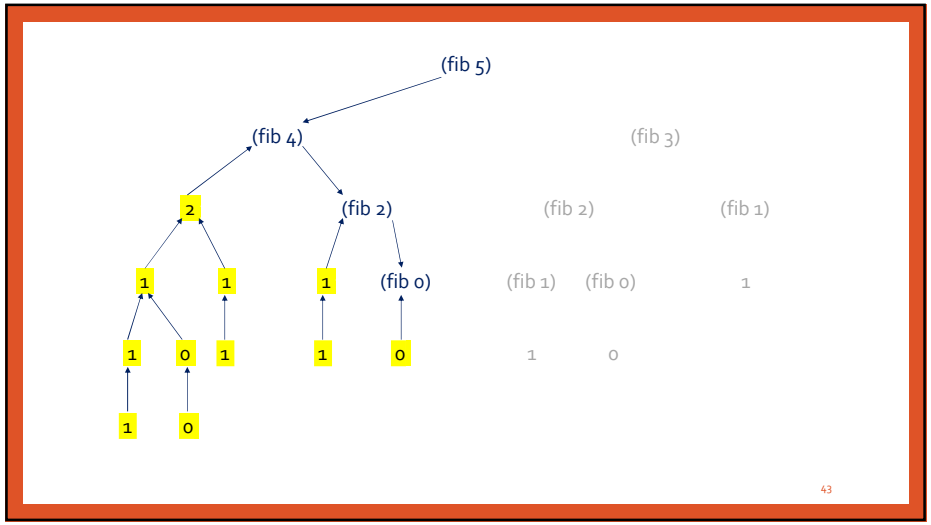
40


(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0
41

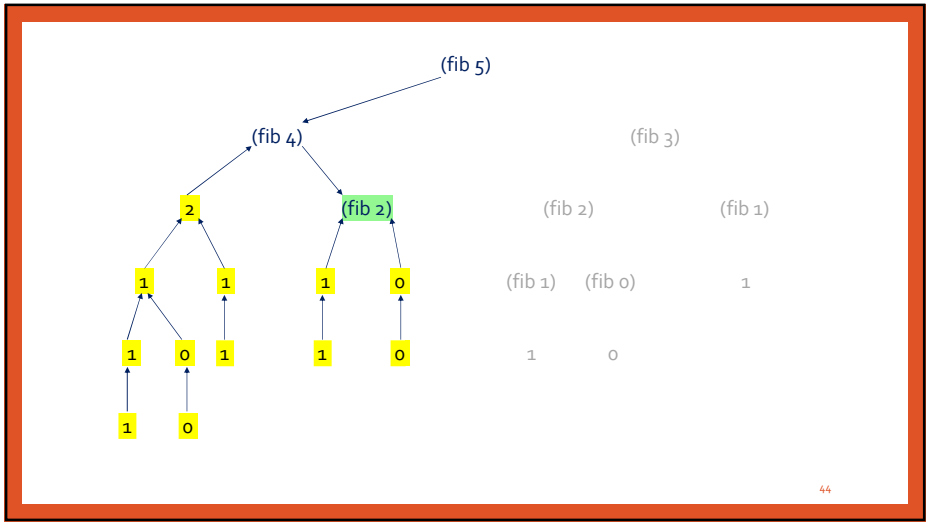
41


(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
1
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0
42

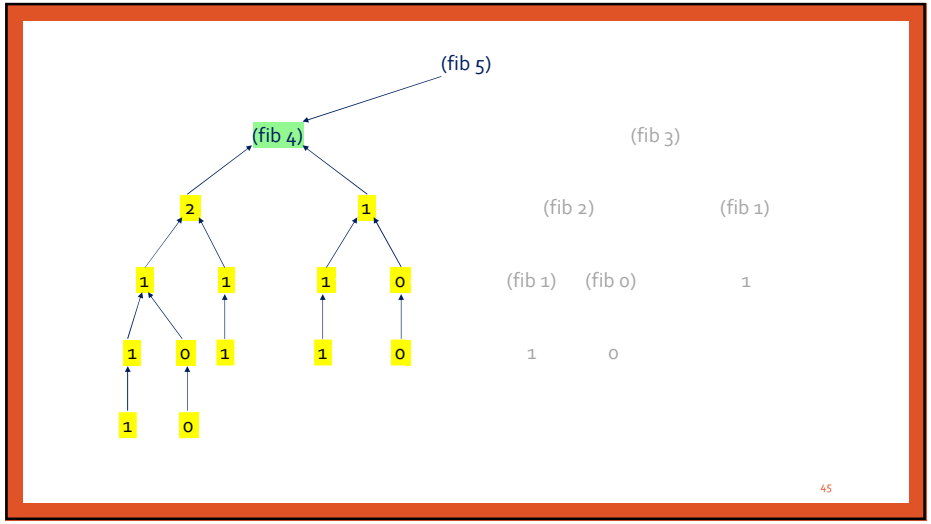
42


(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
1
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
43

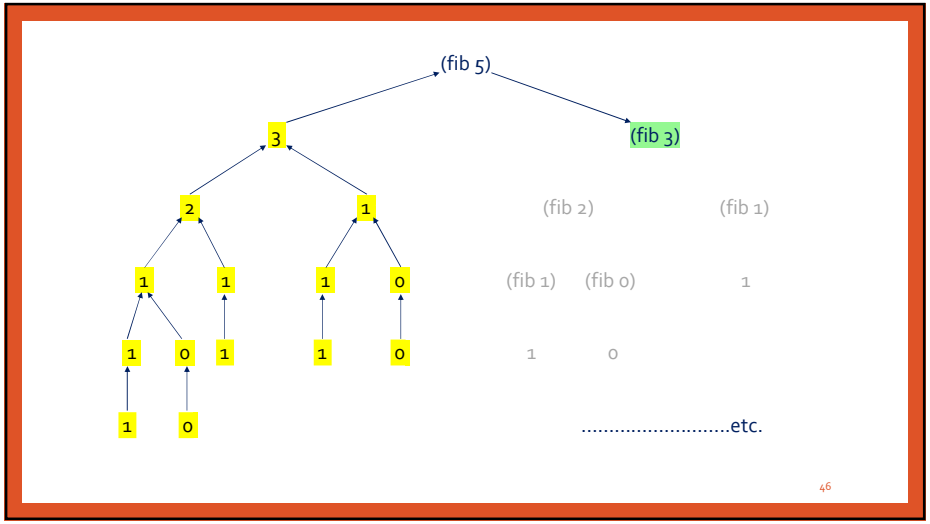
43


(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
1
0
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
44

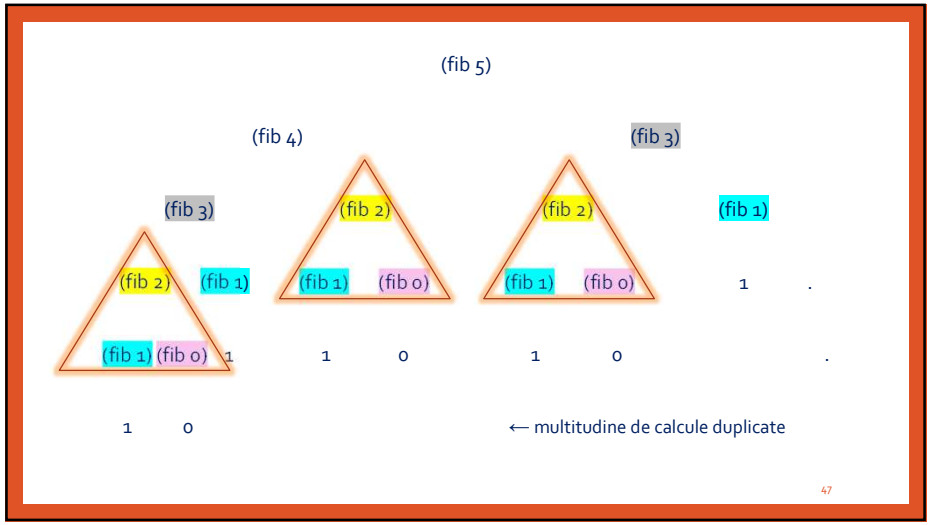
44



45



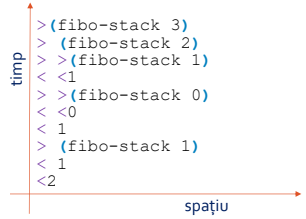
46



47

Observații – recursivitate arborescentă

- **Timp:** Θ(fib(n+1)) (arborele are 2*fib(n+1)-1 noduri)
- **Spațiu:** Θ(n) (stiva la un moment dat reprezintă o singură cale în arbore)
- **Calcul:** realizat integral la revenirea din recursivitate



48

Fibonacci cu recursivitate pe coadă

```
1.  (define (fibo-tail n)
2.    (fibo-tail-helper n 0 1))
3.
4.  (define (fibo-tail-helper n a b)
5.    (if (zero? n)
6.        a
7.        (fibo-tail-helper (- n 1) b (+ a b))))
```

timp = Θ(n)

```
>(fibo-tail-helper 3 0 1)      ← fib(0), fib(1)
>(fibo-tail-helper 2 1 1)      ← fib(1), fib(2)
>(fibo-tail-helper 1 1 2)      ← fib(2), fib(3)
>(fibo-tail-helper 0 2 3)      ← fib(3), fib(4)
>2
```

spațiu = Θ(1)

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă

Comparație

- **Recursivitate pe stivă:** ineficientă spațial din cauza memoriei ocupată de stivă
- **Recursivitate pe coadă:** eficientă spațial și (în general și) temporal
- **Recursivitate arborescentă:** ineficientă spațial din cauza stivei și temporal atunci când aceleași date sunt prelucrate de mai multe noduri din arbore

Atunci când există o soluție iterativă eficientă pentru problemă, aceasta se poate transpune în recursivitate pe coadă. Funcția rezultată va fi:

- Recursivă din punct de vedere textual (funcția se apelează pe ea însăși)
- Iterativă din punct de vedere al procesului generat la execuție

- Mai puțin elegantă decât variantele pe stivă / arborescentă care derivă direct din specificația formală

Cum recunoaștem tipul de recursivitate?

După:

- **numărul de apeluri** recursive pe care un apel le lansează
  - două sau mai multe apeluri → recursivitate arborescentă (care, implicit, folosește și stiva)
  - un singur apel → recursivitate pe stivă sau pe coadă
  - dacă fiecare ramură a unei expresii condiționale lansează maxim un apel recursiv, apelul părinte va lansa maxim un apel recursiv și recursivitatea va fi pe stivă sau pe coadă, nu arborescentă
- **poziția apelurilor** recursive în expresia care descrie valoarea de retur
  - singurul apel recursiv e în poziție finală (valoarea sa este valoarea de retur) → recursivitate pe coadă (condiția trebuie să fie îndeplinită de fiecare ramură a unei expresii condiționale)
  - există un apel care nu e în poziție finală → recursivitate pe stivă (sau arborescentă)

Exemple

Ce tip de recursivitate au funcțiile f și g de mai jos?

```
1.  (define (f x)
2.    (cond ((zero? x)    0)
3.          ((even? x)    (f (/ x 2)))
4.          (else         (+ 1 (f (- x 1))))))
5.
6.  (define (g L result)
7.    (cond ((null? L)          result)
8.          ((list? (car L))    (g (cdr L) (append (g (car L) '()) result)))
9.          (else               (g (cdr L) (cons (car L) result)))))
```

53

53

Exemple

Ce tip de recursivitate au funcțiile f și g de mai jos?

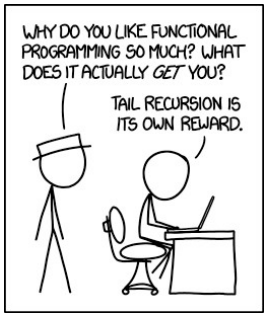
```
1.  (define (f x)                                          pe stivă
2.    (cond ((zero? x)    0)
3.          ((even? x)    (f (/ x 2)))
4.          (else         (+ 1 (f (- x 1)))))             Existența unei variabile de tip acumulator nu înseamnă că avem recursivitate pe coadă!
5.
6.  (define (g L result)                                   arborescentă
7.    (cond ((null? L)          result)
8.          ((list? (car L))    (g (cdr L) (append (g (car L) '()) result)))
9.          (else               (g (cdr L) (cons (car L) result)))))
```

54

54

Tipuri de recursivitate – Cuprins

- Importanța recursivității în paradigma funcțională
- Recursivitate pe stivă
- Recursivitate pe coadă
- Recursivitate arborescentă
- Comparație între tipurile de recursivitate
- Transformarea în recursivitate pe coadă



55

55

Transformarea în recursivitate pe coadă

```
(define (fact-stack n)
  (if (zero? n)
      1
      (* n
         (fact-stack (- n 1)))))
```

```
(define (fact-tail n)
  (fact-tail-helper n 1))

(define (fact-tail-helper n fact)
  (if (zero? n)
      fact
      (fact-tail-helper (- n 1)
                        (* n fact))))
```

- Helper-ul are un argument în plus: acumulatorul în care construim rezultatul pe avansul în recursivitate
- Calculul la care urma să participe rezultatul apelului recursiv este calculul la care participă acumulatorul
- La ieșirea din recursivitate, valoarea de retur nu mai este valoarea pe cazul de bază, ci acumulatorul
- Valoarea funcției pe cazul de bază corespunde adesea valorii inițiale a acumulatorului

56

56

Când acumulatorul este o listă

```
(define (get-odd-stack L)
  (cond
    ((null? L) '())
    ((even? (car L)) (get-odd-stack (cdr L)))
    (else (cons (car L) (get-odd-stack (cdr L))))))
```

```
>(get-odd-stack '(1 4 6 3))
>(cons 1 (get-odd-stack '(4 6 3)))
>(cons 1 (get-odd-stack '(6 3)))
>(cons 1 (get-odd-stack '(3)))
>(cons 1 (cons 3 (get-odd-stack '())))
>(cons 1 (cons 3 '()))
>(cons 1 '(3))
>'(1 3)
```

```
(define (get-odd-tail L acc)
  (cond
    ((null? L) acc)
    ((even? (car L)) (get-odd-tail (cdr L) acc))
    (else (get-odd-tail (cdr L) (cons (car L) acc)))))
```

```
>(get-odd-tail '(1 4 6 3) '())
>(get-odd-tail '(4 6 3) '(1))
>(get-odd-tail '(6 3) '(1))
>(get-odd-tail '(3) '(1))
>(get-odd-tail '() '(3 1))
>'(3 1)
```

ordine inversă!

57

57

Când acumulatorul este o listă

Soluții pentru conservarea ordinii

- Inversarea acumulatorului înainte de retur (pe cazul de bază)

```
... (cond ((null? L) (reverse acc)) ...
```

- Adăugarea fiecărui nou element la sfârșitul acumulatorului (cu append în loc de cons)

```
... (else (get-odd-tail (cdr L) (append acc (list (car L))))) ...
```

Complexitate

- Inversare: Θ(n) (dată de complexitatea lui reverse)
- append în loc de cons: Θ(n²) (Θ(length(acc)) pentru fiecare append în parte)
  (0 + 1 + 2 + ... + (n-1))

58

58

Complexitate `reverse` și `append`

```
1.  (define (reverse L) (rev L '()))
2.  (define (rev L acc)
3.    (if (null? L)
4.        acc
5.        (rev (cdr L) (cons (car L) acc))))        → Θ(length(L))
6.
7.  (define (append A B)
8.    (if (null? A)
9.        B
10.       (cons (car A) (append (cdr A) B))))       → Θ(length(A))
```

Concluzie: Pentru eficiență folosim inversarea acumulatorului la final, nu adăugarea fiecărui element la sfârșit.

59

59

Calcul Lambda

60

60

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ-expresie
- Statutul variabilelor într-o λ-expresie
- Evaluarea unei λ-expresii
- Forma normală a unei λ-expresii

61

61

Memento: λ-expresia

Sintaxa

e ≡ **x** variabilă
| **λx.e1** funcție (unară, anonimă) cu parametrul formal x și corpul e
| **(e1 e2)** aplicație a expresiei e1 asupra parametrului efectiv e2

Semantica (Modelul substituției)

Pentru a evalua **(λx.e1 e2)** (funcția cu parametrul formal x și corpul e1, aplicată pe e2):
- Peste tot în e1, identificatorul x este înlocuit cu e2
- Se evaluează noul corp e1 și se întoarce rezultatul (se notează $e1_{[e2/x]}$)

62

62

Aparițiile unei variabile într-o λ-expresie

$$\lambda x.(x\ \lambda y.x)$$

Variabila x: 3 apariții conform cărora putem rescrie expresia ca $\lambda x_1.(x_2\ \lambda y.x_3)$

Variabila y: 1 apariție conform căreia putem rescrie expresia ca $\lambda x.(x\ \lambda y_1.x)$

Vom distinge între:
- variabilele al căror nume nu contează (și ar putea fi oricare altul)
  și
- variabilele ar căror nume este un alias pentru valori din exteriorul expresiei

63

63

Apariții legate / libere într-o expresie

Apariția x_n este **legată** în E dacă:
- **E = …λx_n.e…**
- **E = …λx.e… și x_n apare în e**

Variabila de după **λ** = variabila de legare
Apariția de după **λ** = apariția care leagă (restul aparițiilor lui x în corpul e)

Altfel, apariția x_n este **liberă** în E.

$$\lambda x_1.(x_2\ \lambda y.x_3)$$

legată (apariția care leagă) — legată (aparițiile în corpul funcției de parametru x) — legată

64

64

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)
(y λy.x)
λz.((+ z) x)
(λx. λy.(x y) y)
λx.(y λy.(x (y z)))
(x λx.(λx.y λy.(x z)))

65

65

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)
(y λy.x)
λz.((+ z) x)
(λx. λy.(x y) y)
λx.(y λy.(x (y z)))
(x λx.(λx.y λy.(x z)))

66

66

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)
(y λy.x)
λz.((+ z) x)
(λx. λy.(x y) y)
λx.(y λy.(x (y z)))
(x λx.(λx.y λy.(x z)))

67

67

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)
(y λy.x)
λz.((+ z) x)
(λx. λy.(x y) y)
λx.(y λy.(x (y z)))
(x λx.(λx.y λy.(x z)))

68

68

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

69

69

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

70

70

Exemple de apariții legate / libere

Vom marca aparițiile **legate** ale fiecărei variabile cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

71

71

Observații

- O apariție este legată sau liberă **într-o expresie**

$$E = \lambda x.(y\ \lambda y.(x\ (y\ z))) =_{\text{notație}} \lambda x.e$$

legată în E

liberă în e

$$e = (y\ \lambda y.(x\ (y\ z)))$$

- Numele aparițiilor legate ale unei variabile nu contează (le putem redenumi pe toate cu un același nou identificator, semnificația expresiei rămânând aceeași)

$$E = \lambda x.(y\ \lambda y.(x\ (y\ z))) = \lambda a.(y\ \lambda b.(a\ (b\ z)))$$

(valoare externă lui E) acest y nu are nicio legătură cu acest y (parametrul funcției interne)

72

72

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ-expresie
- Statutul variabilelor într-o λ-expresie
- Evaluarea unei λ-expresii
- Forma normală a unei λ-expresii

73

73

Variabile legate / libere într-o expresie

Variabila x este **legată** în E dacă **toate aparițiile lui x sunt legate în E**.
Altfel, variabila x este **liberă** în E.

Exemplu: (x λx.x) din punct de vedere al statutului aparițiilor devine
(x λx.x) din punct de vedere al statutului variabilelor („din cauza" primului x).

Observații
- Ca și în cazul aparițiilor, o variabilă este legată sau liberă **într-o expresie**
- Ca și în cazul aparițiilor, **numele variabilelor legate nu contează**

74

74

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)
(y λy.x)
λz.((+ z) x)
(λx. λy.(x y) y)
λx.(y λy.(x (y z)))
(x λx.(λx.y λy.(x z)))

75

75

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)
(y λy.x)
λz.((+ z) x)
(λx. λy.(x y) y)
λx.(y λy.(x (y z)))
(x λx.(λx.y λy.(x z)))

76

76

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

77

77

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

78

78

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y) dar dacă ne limităm la subexpresia λy.(x y) statutul variabilelor se inversează!

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

79

79

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

80

80

Exemple de variabile legate / libere

Vom marca variabilele **legate** cu **portocaliu** și pe cele **libere** cu **verde**.

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

λx.(y λy.(x (y z)))

(x λx.(λx.y λy.(x z)))

81

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ-expresie
- Statutul variabilelor într-o λ-expresie
- Evaluarea unei λ-expresii
- Forma normală a unei λ-expresii

82

β-redex și β-reducere

Memento: Semantica λ-expresiilor (Modelul substituției)

Pentru a evalua **(λx.e1 e2)** (funcția cu parametrul formal x și corpul e1, aplicată pe e2):

- ~~Peste tot în e1, identificatorul x este înlocuit cu e2~~
- Peste tot în e1, aparițiile libere ale lui x (libere în e1 !) sunt înlocuite cu e2 (nu are sens să înlocuiesc și aparițiile legate, întrucât acestea se numesc tot x doar întâmplător)
- Se evaluează noul corp e1 și se întoarce rezultatul (se notează $e1_{[e2/x]}$)

β-redex = λ-expresie de forma $(\lambda x.e_1\ e_2)$

β-reducere = efectuarea calculului $(\lambda x.e_1\ e_2) \rightarrow_\beta e_{1[e_2/x]}$

83

Greșeli apărute la β-reducere

Exemplu: $(\lambda x.\lambda y.(\underbrace{x\ y}_{e1})\ \underbrace{y}_{e2}) \rightarrow_\beta \lambda y.(x\ y)_{[y/x]} = \lambda y.(y\ y)$ (greșit!)

Trebuia să obținem

o funcție care îl aplică pe y asupra argumentului său

$\neq$

Am obținut

o funcție care își aplică argumentul asupra lui însuși

Ce a mers rău?

În urma înlocuirii, apariția lui y (care era liberă în e2) s-a trezit legată în e1 (la un argument cu care nu avea nicio legătură).

Generalizare

Conflict de nume între variabilele legate din e1 și variabilele libere din e2 → greșeli în evaluare

84

Soluția: α-conversia

α-conversie = redenumirea variabilelor legate din corpul unei funcții **λx.e1** a.î. ele să nu coincidă cu variabilele libere din parametrul efectiv **e2** pe care aplicăm funcția

Observații
- Numele variabilelor legate oricum nu contează, deci ele pot fi redenumite
- Noul nume trebuie să nu intre în conflict cu variabilele libere din e1 și din e2

Exemplu: $(\lambda x.\lambda y.(x\ y)\ y) \rightarrow_\alpha (\lambda x.\lambda z.(x\ z)\ y) \rightarrow_\beta \lambda z.(x\ z)_{[y/x]} = \lambda z.(y\ z)$ (corect!)

($e1$: $\lambda y.(x\ y)$, $e2$: y)

85

85

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y)

86

86

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y)

87

87

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y)

88

88

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y) $\rightarrow_\alpha$
((λx.λy.λt.(x (y t)) z) y)

89

89

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y) $\rightarrow_\alpha$
((λx.λy.λt.(x (y t)) z) y) $\rightarrow_\beta$
(λy.λt.(z (y t)) y)

90

90

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y) $\rightarrow_\alpha$
((λx.λy.λt.(x (y t)) z) y) $\rightarrow_\beta$
(λy.λt.(z (y t)) y)

91

91

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y) $\rightarrow_\alpha$
((λx.λy.λt.(x (y t)) z) y) $\rightarrow_\beta$
(λy.λt.(z (y t)) y)

92

92

Exemplu de secvență de reducere

((λx.λy.λz.(x (y z)) z) y) $\rightarrow_\alpha$
((λx.λy.λt.(x (y t)) z) y) $\rightarrow_\beta$
(λy.λt.(z (y t)) y) $\rightarrow_\beta$
λt.(z (y t))

93

93

Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o λ-expresie
- Statutul variabilelor într-o λ-expresie
- Evaluarea unei λ-expresii
- Forma normală a unei λ-expresii

94

94

Forma normală a unei λ-expresii

λ-expresie în **forma normală** ⇔ λ-expresie care nu conține niciun β-redex

Întrebări
- Are orice λ-expresie o formă normală?
- Forma normală este unică? (sau secvențe distincte de reducere pot duce la forme normale distincte?)
- Dacă o λ-expresie admite o formă normală, se poate garanta găsirea ei?

Observație (ajutătoare pentru întrebările anterioare)
Calculul Lambda este un model de calculabilitate.
λ-expresiile sunt practic programe capabile să ruleze pe o ipotetică Mașină Lambda.

95

95

Are orice λ-expresie o formă normală?

NU.

Exemplu: (λx.(x x) λx.(x x)) $\rightarrow_\beta$ (λx.(x x) λx.(x x)) $\rightarrow_\beta$ (λx.(x x) λx.(x x)) $\rightarrow_\beta$ ...

λ-expresie **reductibilă** ⇔ admite o secvență finită de reducere până la o formă normală
Altfel, λ-expresia este **ireductibilă**.

96

96

Forma normală este unică?

DA.

Teorema Church-Rosser

Dacă { e $\rightarrow^*$ a ; și ; e $\rightarrow^*$ b } atunci $\exists$ d a.î. { a $\rightarrow^*$ d ; și ; b $\rightarrow^*$ d } ($\rightarrow^*$ = secvență de reducere)

Explicație: Dacă a și b ar fi forme normale distincte, prin definiție a și b nemaiconținând niciun β-redex, ele nu s-ar putea reduce suplimentar către un același d.

97

97

Se poate garanta găsirea formei normale?

DA.

Teorema normalizării

Pentru orice λ-expresie reductibilă, se poate ajunge la forma ei normală aplicând **reducere stânga->dreapta** (reducând mereu cel mai din stânga β-redex, ca la evaluarea normală).

Exemplu: $E_1 = (\lambda x.(x\ x)\ \lambda x.(x\ x))$

$E_2 = (\lambda x.y\ E_1) \rightarrow_\beta y$ ← o reducere dreapta->stânga nu s-ar termina niciodată

Concluzie: Evaluarea aplicativă e mai eficientă, dar evaluarea normală e mai sigură.

98

98

Rezumat

Teza lui Church
Tipuri de recursivitate
Apariții ale unei variabile într-o expresie
Variabile într-o expresie
β-redex și β-reducere
α-conversie
Forma normală
Expresie ireductibilă
Teorema normalizării

99

99

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate
Apariții ale unei variabile într-o expresie
Variabile într-o expresie
β-redex și β-reducere
α-conversie
Forma normală
Expresie ireductibilă
Teorema normalizării

100

100

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie
Variabile într-o expresie
β-redex și β-reducere
α-conversie
Forma normală
Expresie ireductibilă
Teorema normalizării

101

101

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie
β-redex și β-reducere
α-conversie
Forma normală
Expresie ireductibilă
Teorema normalizării

102

102

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)
β-redex și β-reducere
α-conversie
Forma normală
Expresie ireductibilă
Teorema normalizării

103

103

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)
β-redex și β-reducere: $(\lambda x.e_1\ e_2)$, $(\lambda x.e_1\ e_2) \rightarrow_\beta e_{1[e_2/x]}$
α-conversie
Forma normală
Expresie ireductibilă
Teorema normalizării

104

104

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)
β-redex și β-reducere: (λx.e1 e2), (λx.e1 e2) $\rightarrow_\beta$ $e1_{[e2/x]}$
α-conversie: (λx. ... λy.e1 ... e2) $\rightarrow_\alpha$ (λx. ... $\lambda t.e1_{[t/y]}$ ... e2) unde t nu era liberă în e1, e2
Forma normală
Expresie ireductibilă
Teorema normalizării

105

105

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)
β-redex și β-reducere: (λx.e1 e2), (λx.e1 e2) $\rightarrow_\beta$ $e1_{[e2/x]}$
α-conversie: (λx. ... λy.e1 ... e2) $\rightarrow_\alpha$ (λx. ... $\lambda t.e1_{[t/y]}$ ... e2) unde t nu era liberă în e1, e2
Forma normală: λ-expresia nu conține niciun β-redex
Expresie ireductibilă
Teorema normalizării

106

106

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)
β-redex și β-reducere: (λx.e1 e2), (λx.e1 e2) $\rightarrow_\beta$ $e1_{[e2/x]}$
α-conversie: (λx. ... λy.e1 ... e2) $\rightarrow_\alpha$ (λx. ... $\lambda t.e1_{[t/y]}$ ... e2) unde t nu era liberă în e1, e2
Forma normală: λ-expresia nu conține niciun β-redex
Expresie ireductibilă: nu poate fi redusă la o formă normală (altfel – reductibilă)
Teorema normalizării

107

107

Rezumat

Teza lui Church: Orice calcul efectiv poate fi modelat în Calcul Lambda (cu funcții recursive)
Tipuri de recursivitate: pe stivă (ineficient spațial), pe coadă, arborescentă (ineficient spațial/temporal)
Apariții ale unei variabile într-o expresie: legate (λx.e, λx. ...x...), libere (restul)
Variabile într-o expresie: legate (toate aparițiile sunt legate), libere (restul)
β-redex și β-reducere: (λx.e1 e2), (λx.e1 e2) $\rightarrow_\beta$ $e1_{[e2/x]}$
α-conversie: (λx. ... λy.e1 ... e2) $\rightarrow_\alpha$ (λx. ... $\lambda z.e1_{[z/y]}$ ... e2) unde z nu era liberă în e1, e2
Forma normală: λ-expresia nu conține niciun β-redex
Expresie ireductibilă: nu poate fi redusă la o formă normală (altfel – reductibilă)
Teorema normalizării: Reducerea stânga->dreapta garantează găsirea formei normale (când aceasta există)

108

108