

# PARADIGME DE PROGRAMARE

---

## Curs 2

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

# 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

# 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).

# 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

# 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



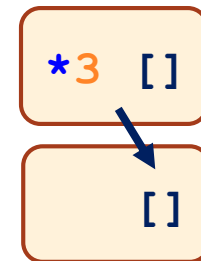
Stiva procesului

# 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))
```

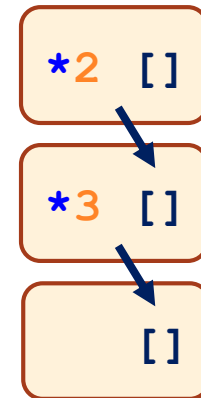


Stiva procesului

# 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)))
```

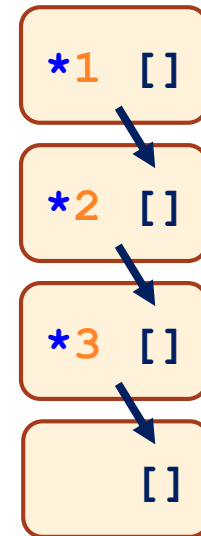


Stiva procesului

# 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))))
```



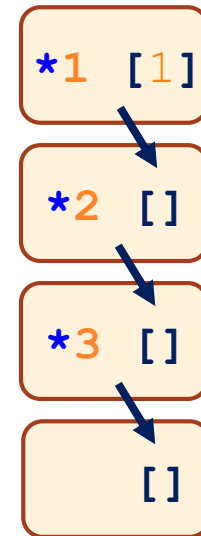
Stiva procesului



# 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)))
```

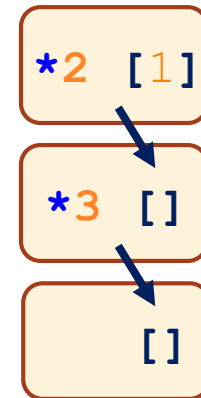


Stiva procesului

# 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))
```

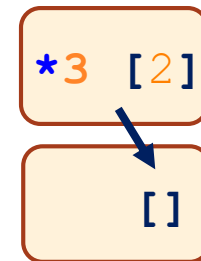


Stiva procesului

# 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

# Exemplu – recursivitate pe stivă

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

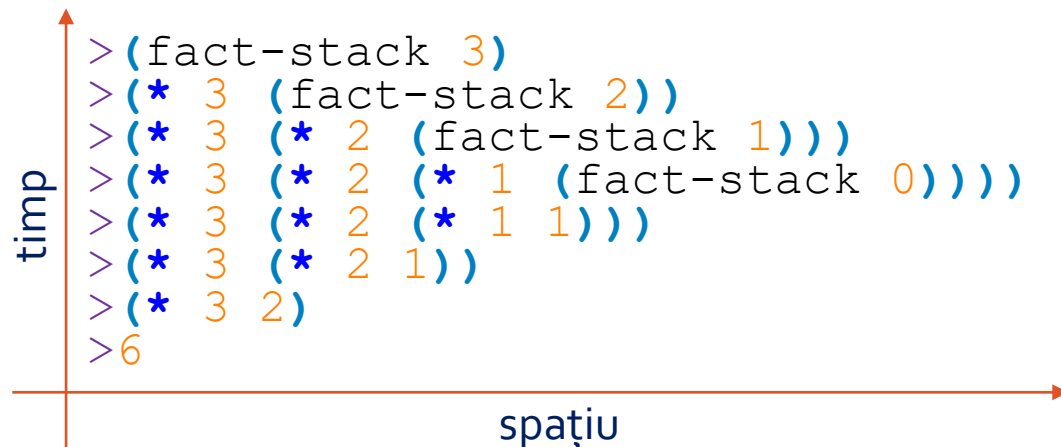
```
> (fact-stack 3)
> (* 3 (fact-stack 2))
> (* 3 (* 2 (fact-stack 1)))
> (* 3 (* 2 (* 1 (fact-stack 0))))
> (* 3 (* 2 (* 1 1)))
> (* 3 (* 2 1))
> (* 3 2)
> 6
```

[6]

Stiva procesului

# Observații – recursivitate pe stivă

- **Timp:**  $\Theta(n)$  (se efectuează  $n$  înmulțiri și stiva se redimensionează de  $2 \cdot n$  ori)
- **Spațiu:**  $\Theta(n)$  (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)



# Comparație cu rezolvarea imperativă

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

- **Timp:**  $\Theta(n)$  (se efectuează  $n$  înmulțiri)
- **Spațiu:**  $\Theta(1)$  (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?

# 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

# 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

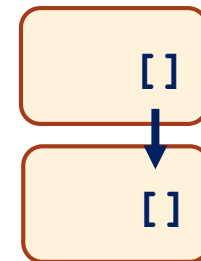


# Exemplu – recursivitate pe coadă

```
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

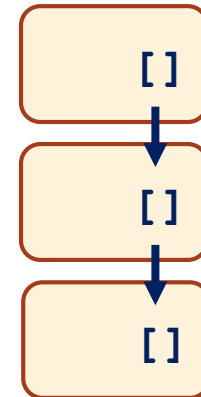
# Exemplu – recursivitate pe coadă

```
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)
```

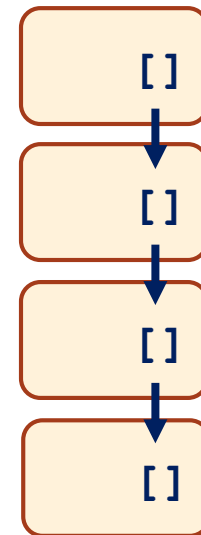


Stiva procesului

# 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)
```



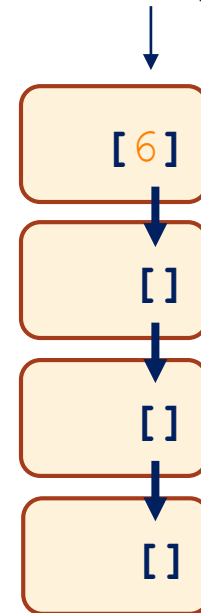
Stiva procesului

# 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

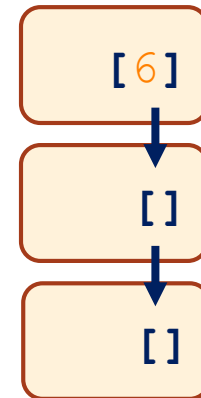


Stiva procesului

# 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
```

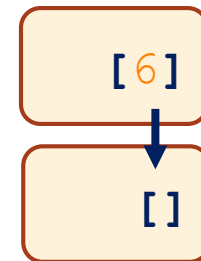


Stiva procesului

# 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

# 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
```

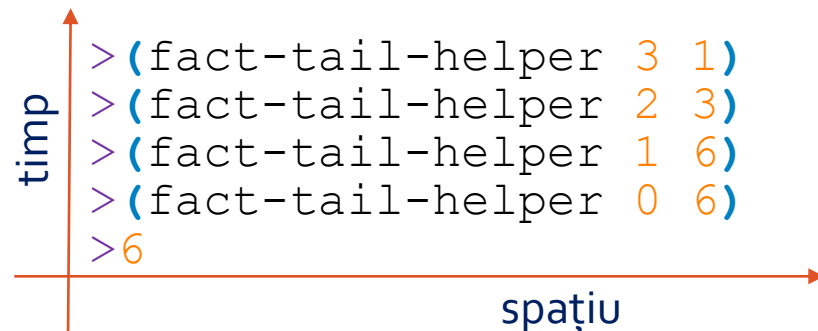
[6]

Stiva procesului

# 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 rețin starea programului)
- **Calcul:** realizat integral pe avansul în recursivitate



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

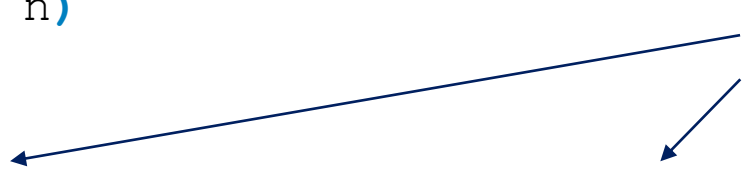


# Tipuri de recursivitate – Cuprins

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

# Exemplu – recursivitate arborescentă

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



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

```
> (fibonacci 3)
> (fibonacci 2)
> (fibonacci 1)
< 1
> (fibonacci 0)
< 0
< 1
> (fibonacci 1)
< 1
< 2
```

← (fibonacci 1) se calculează de 2 ori, iar numărul de calcule redundante crește exponențial cu n

(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

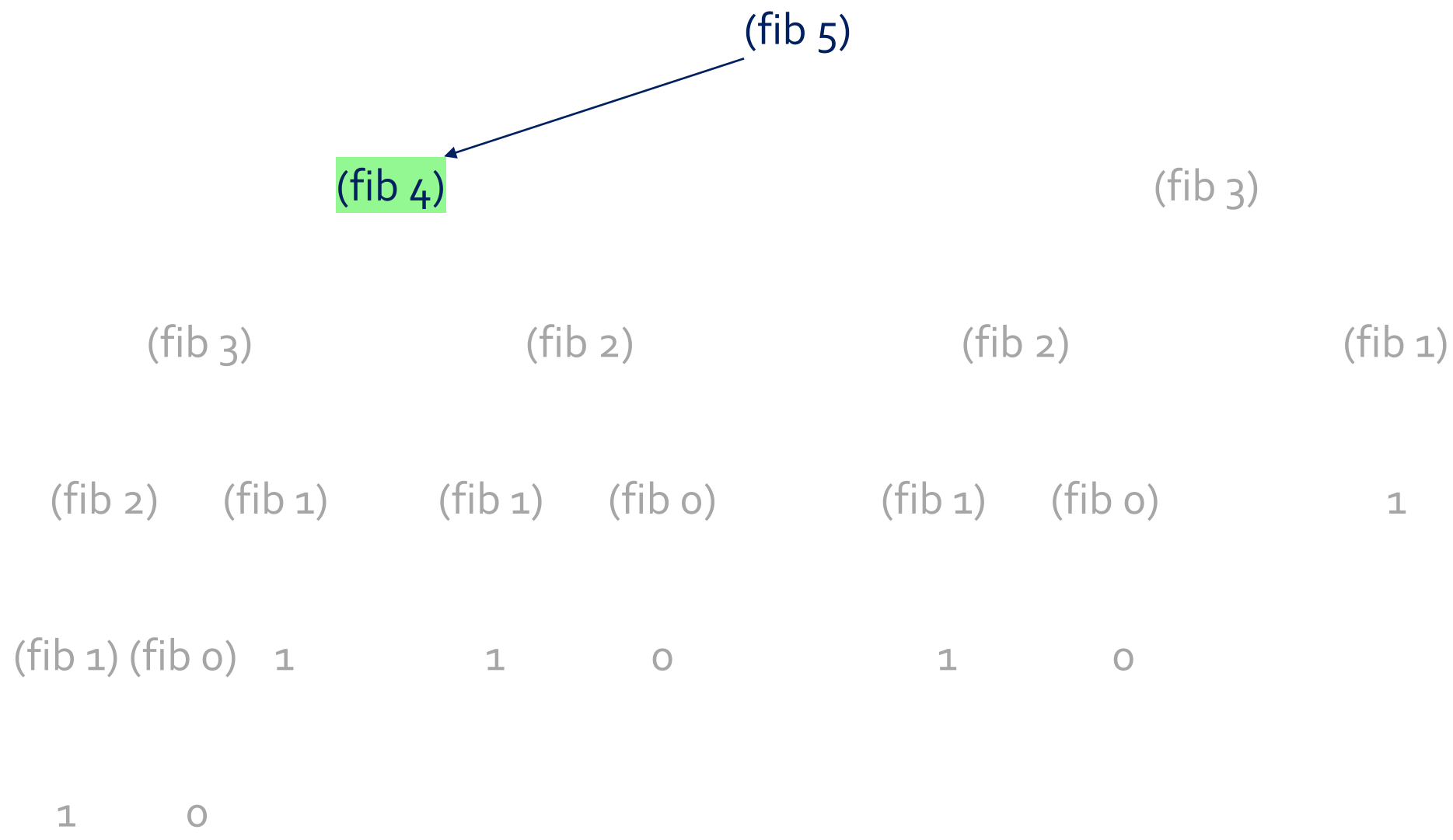
0

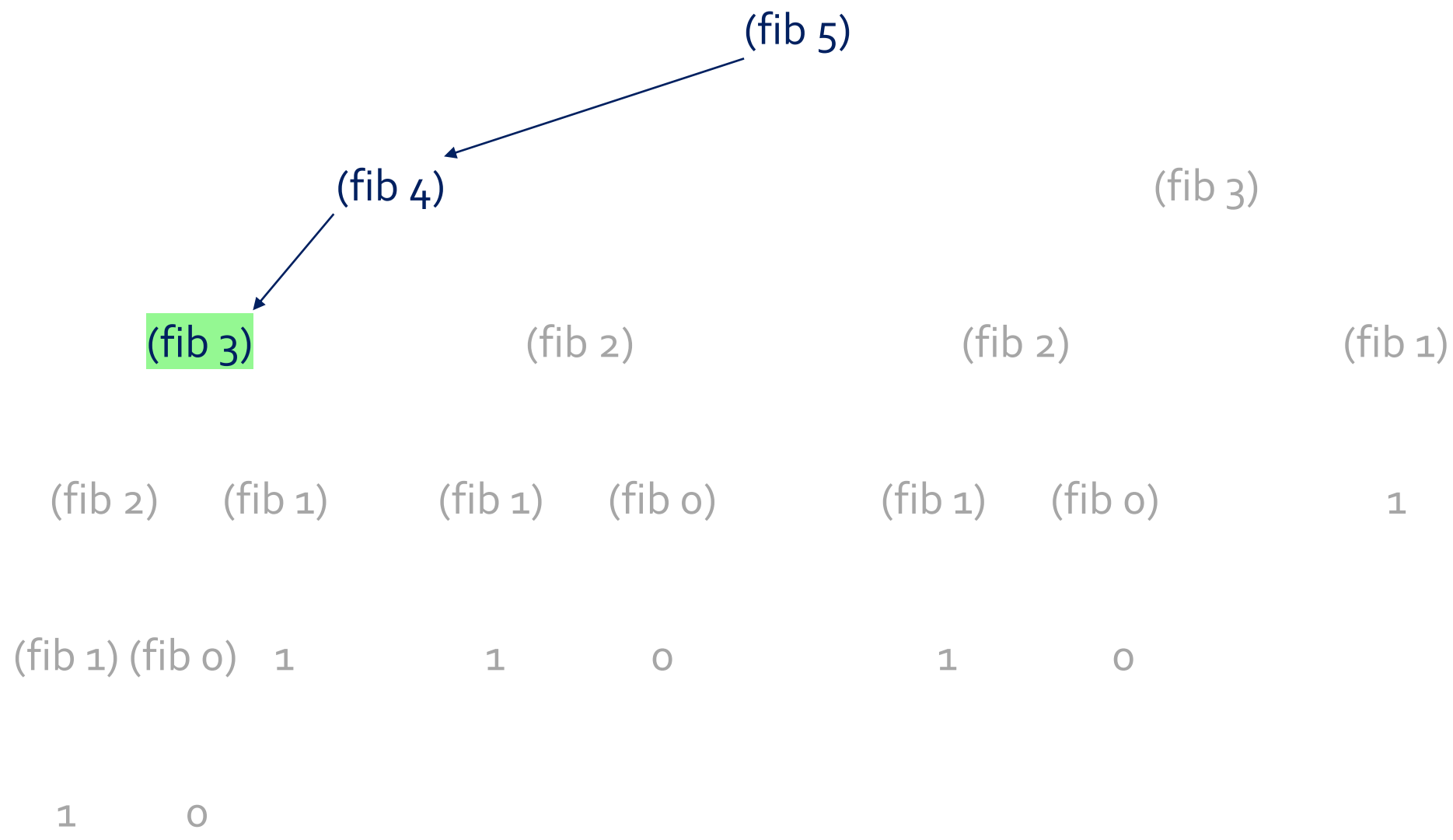
1

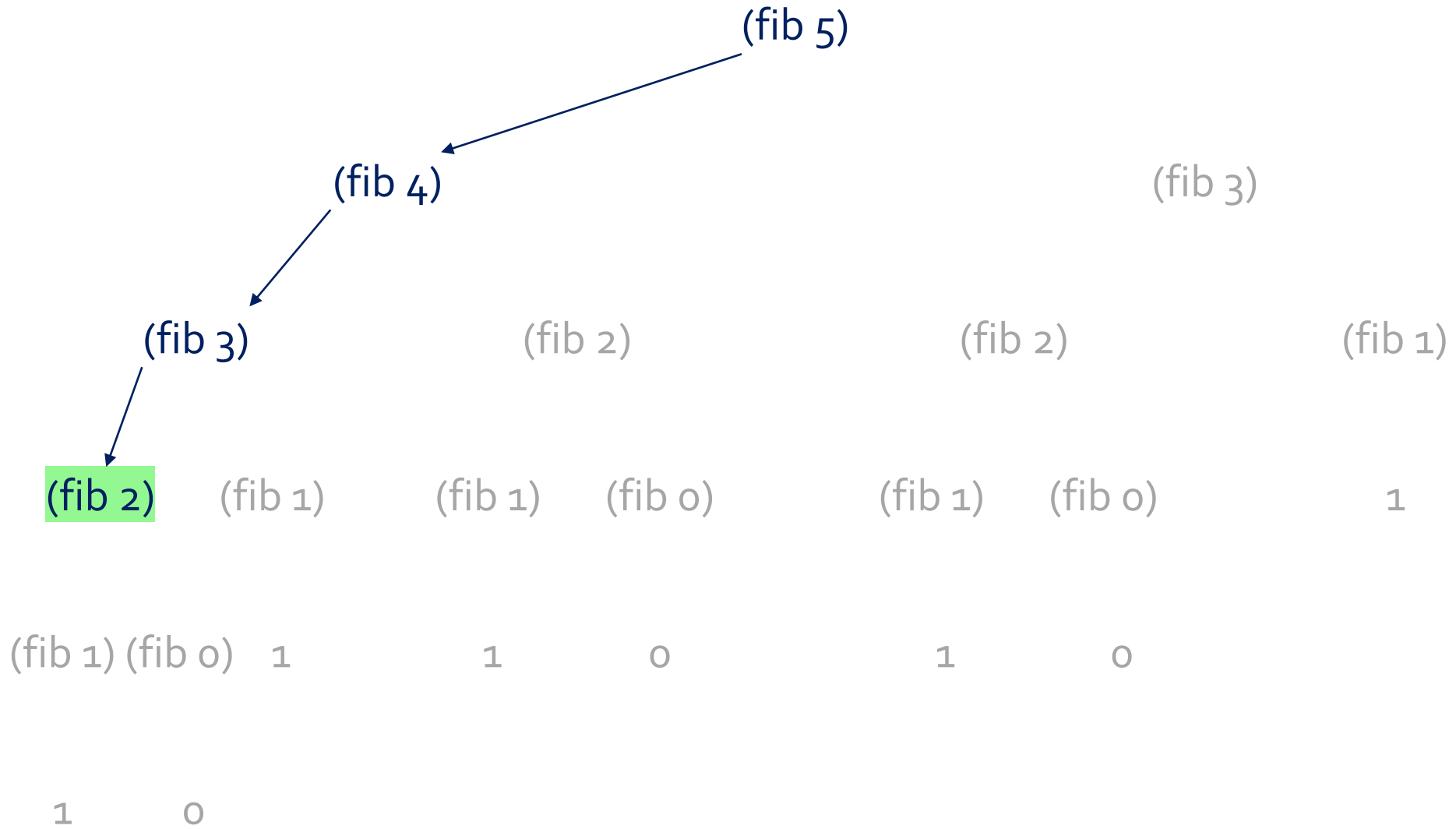
0

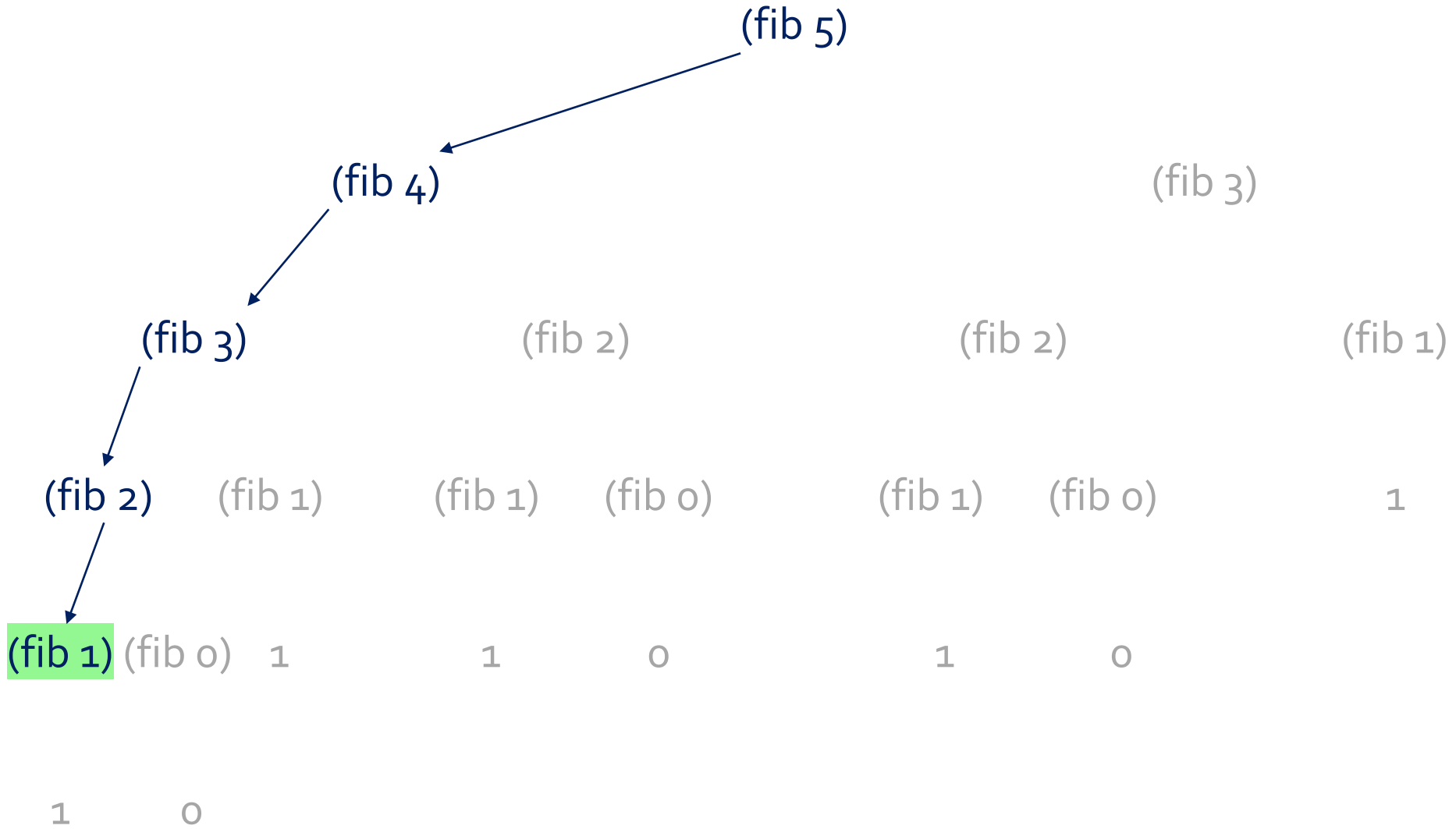
1

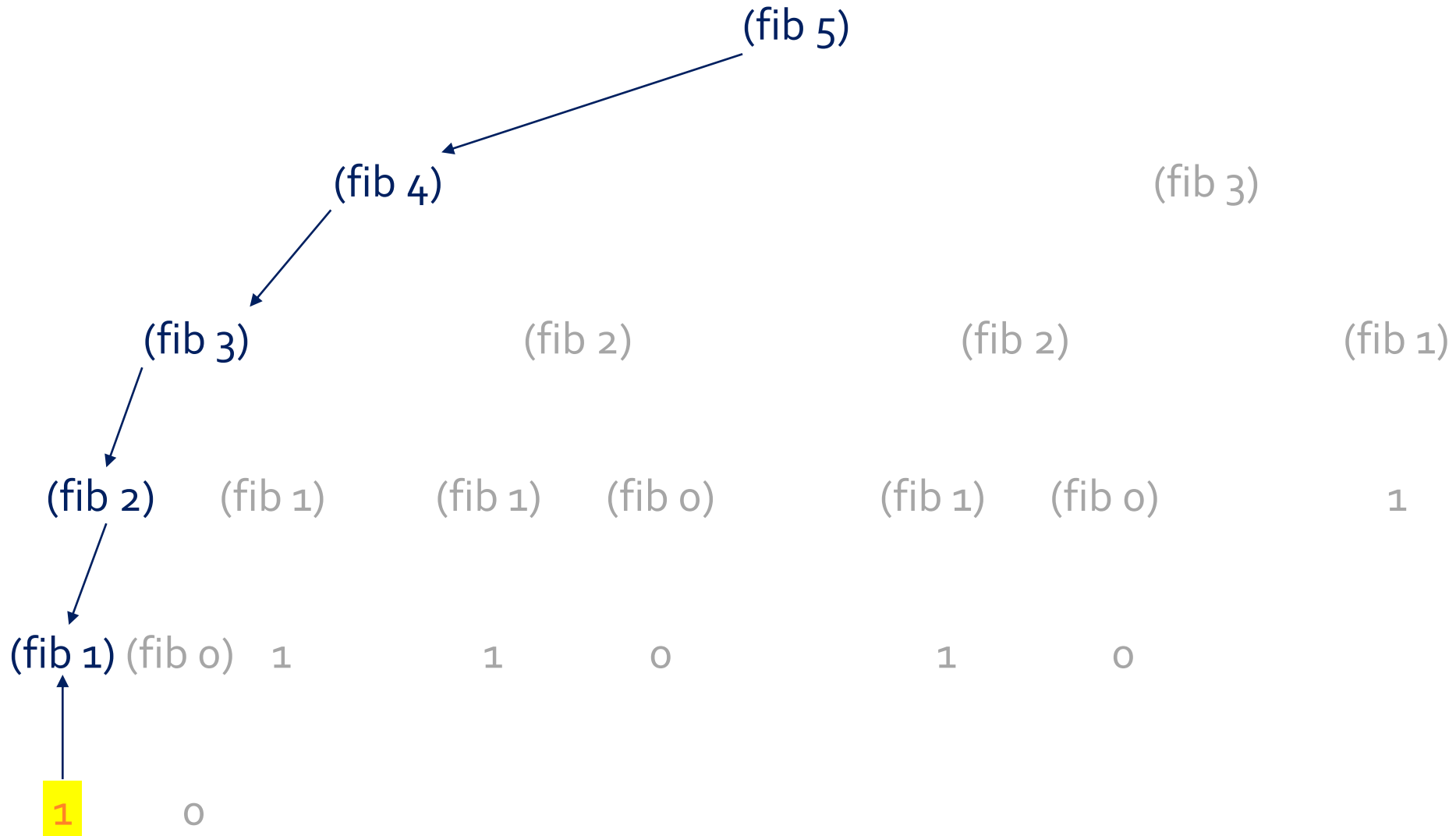
0



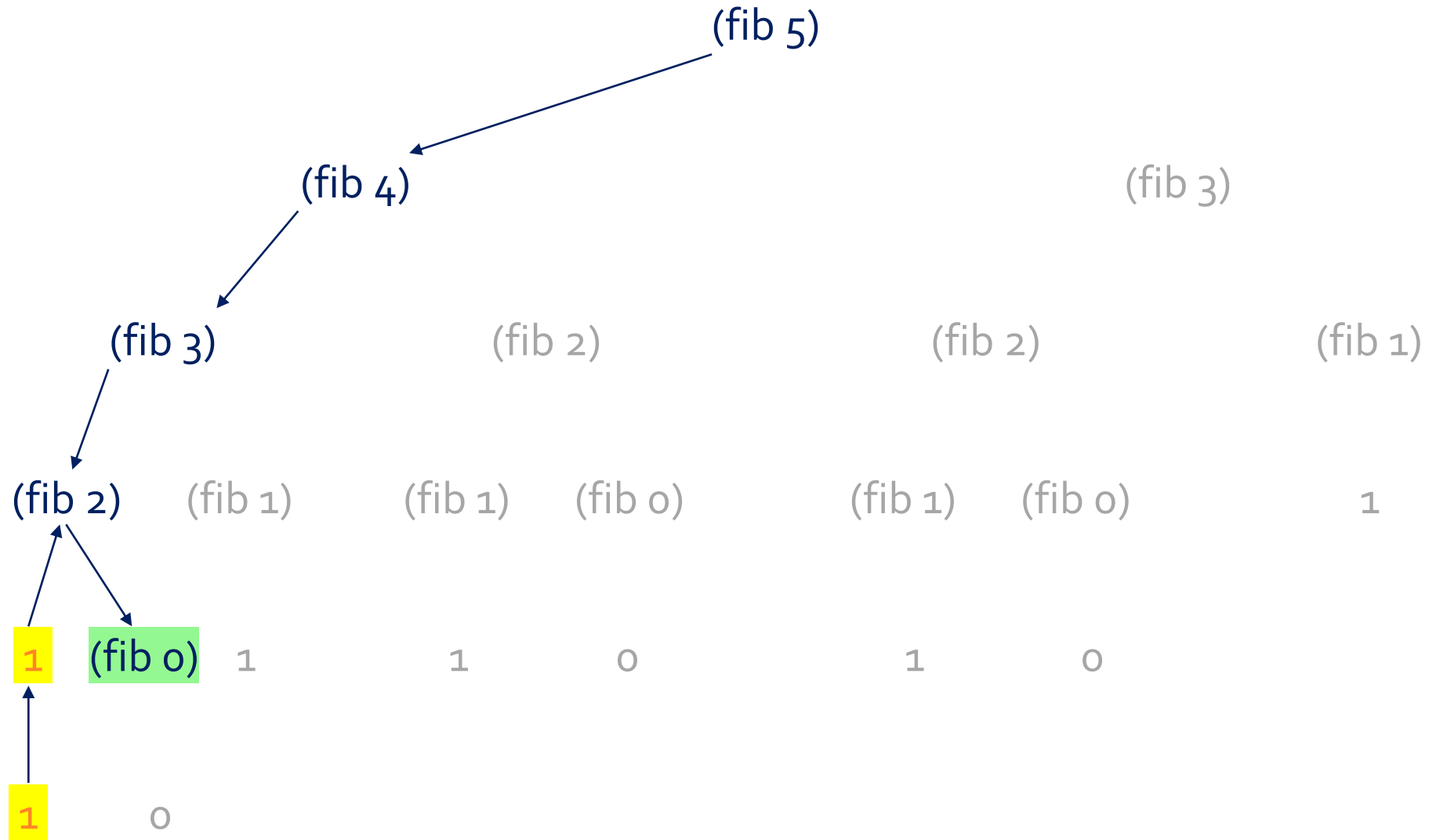


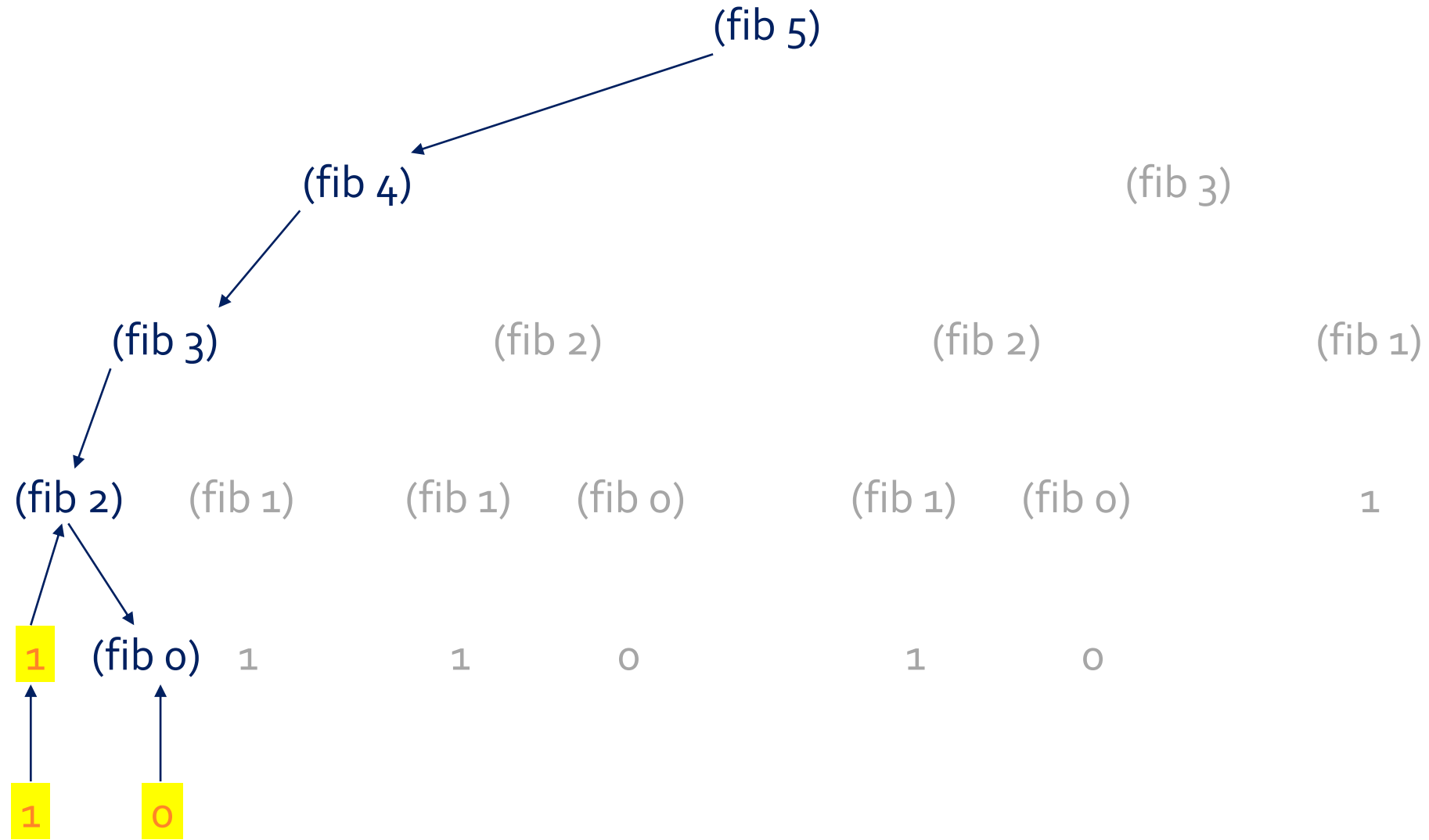


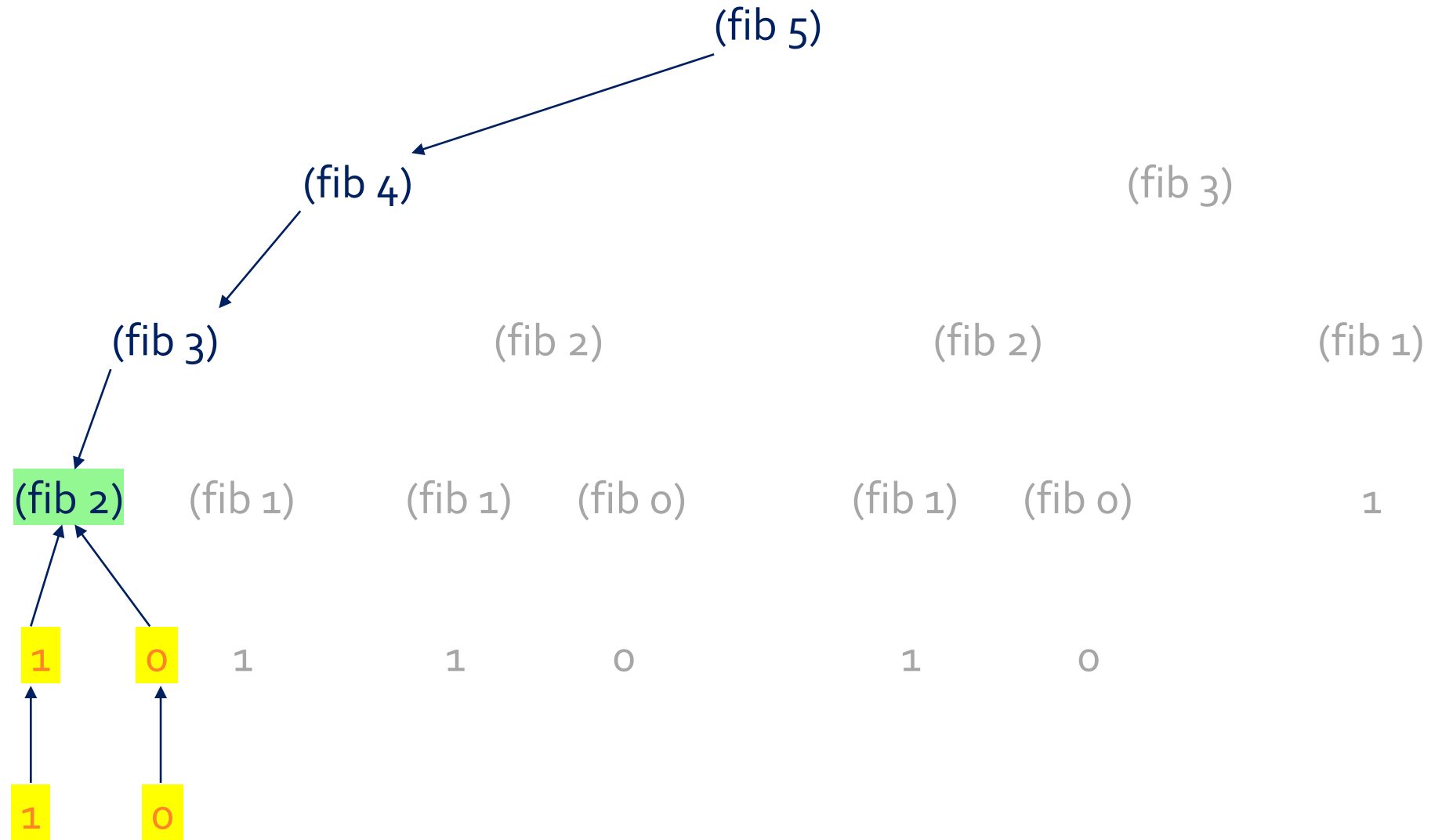


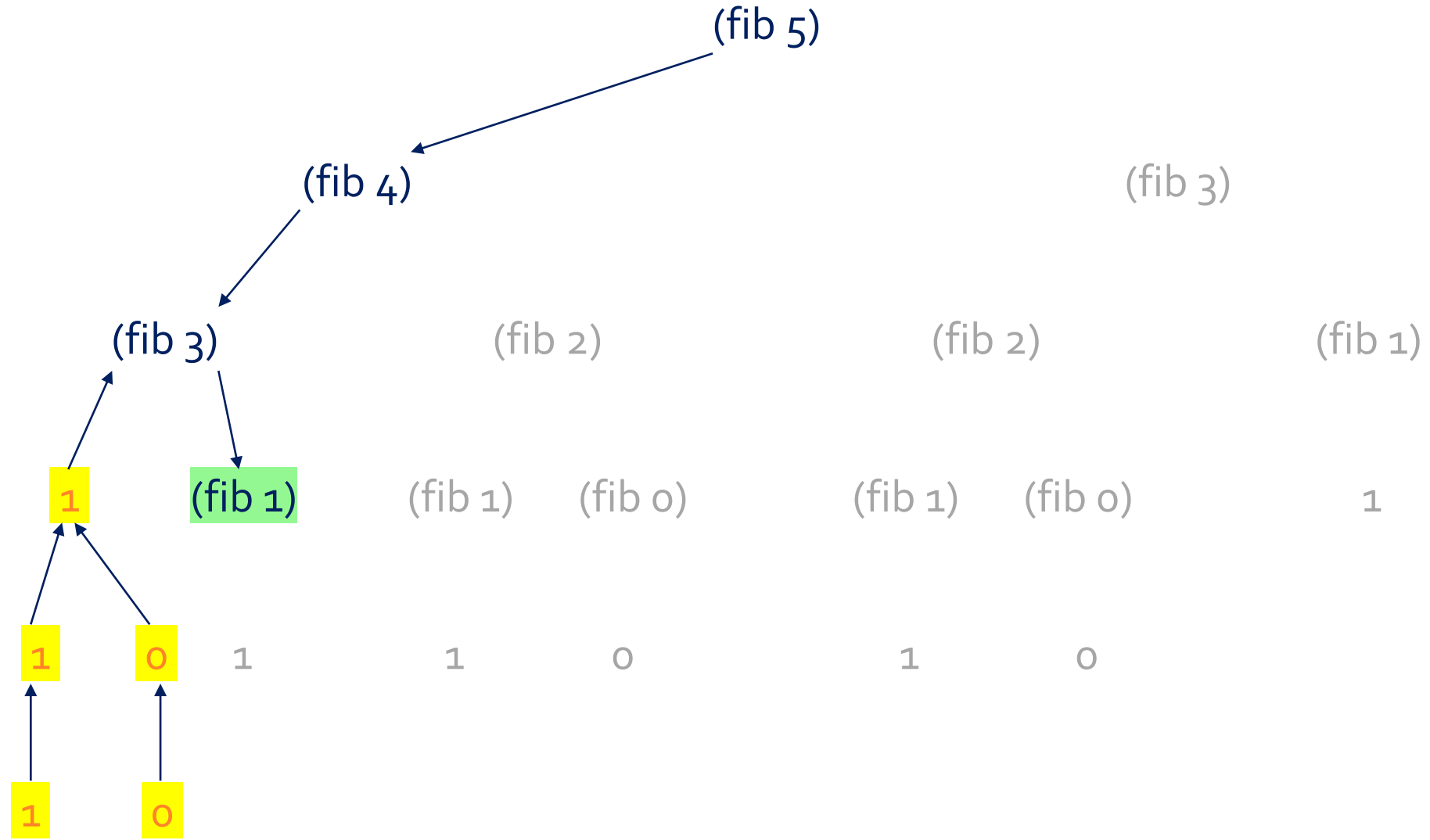


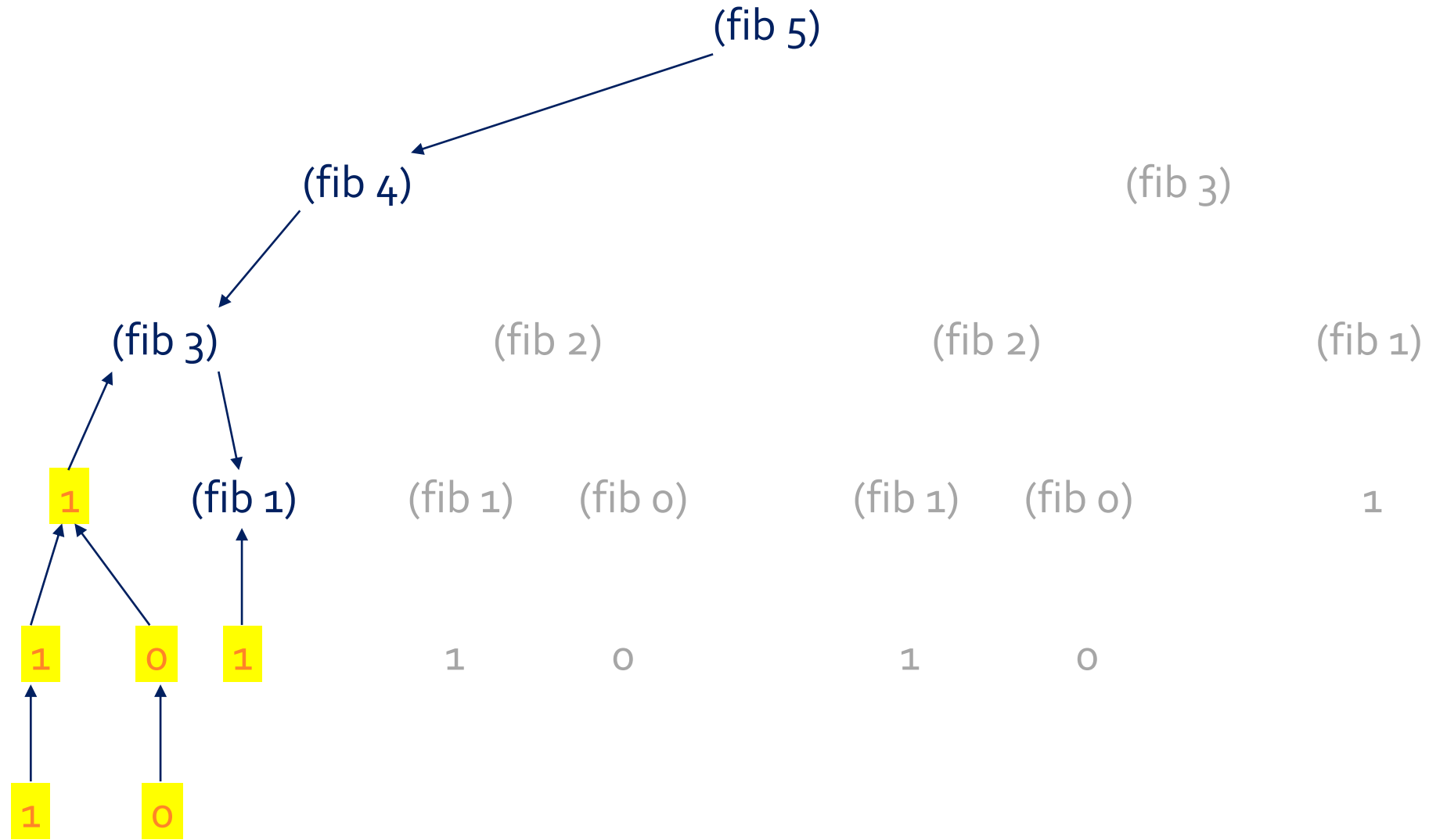


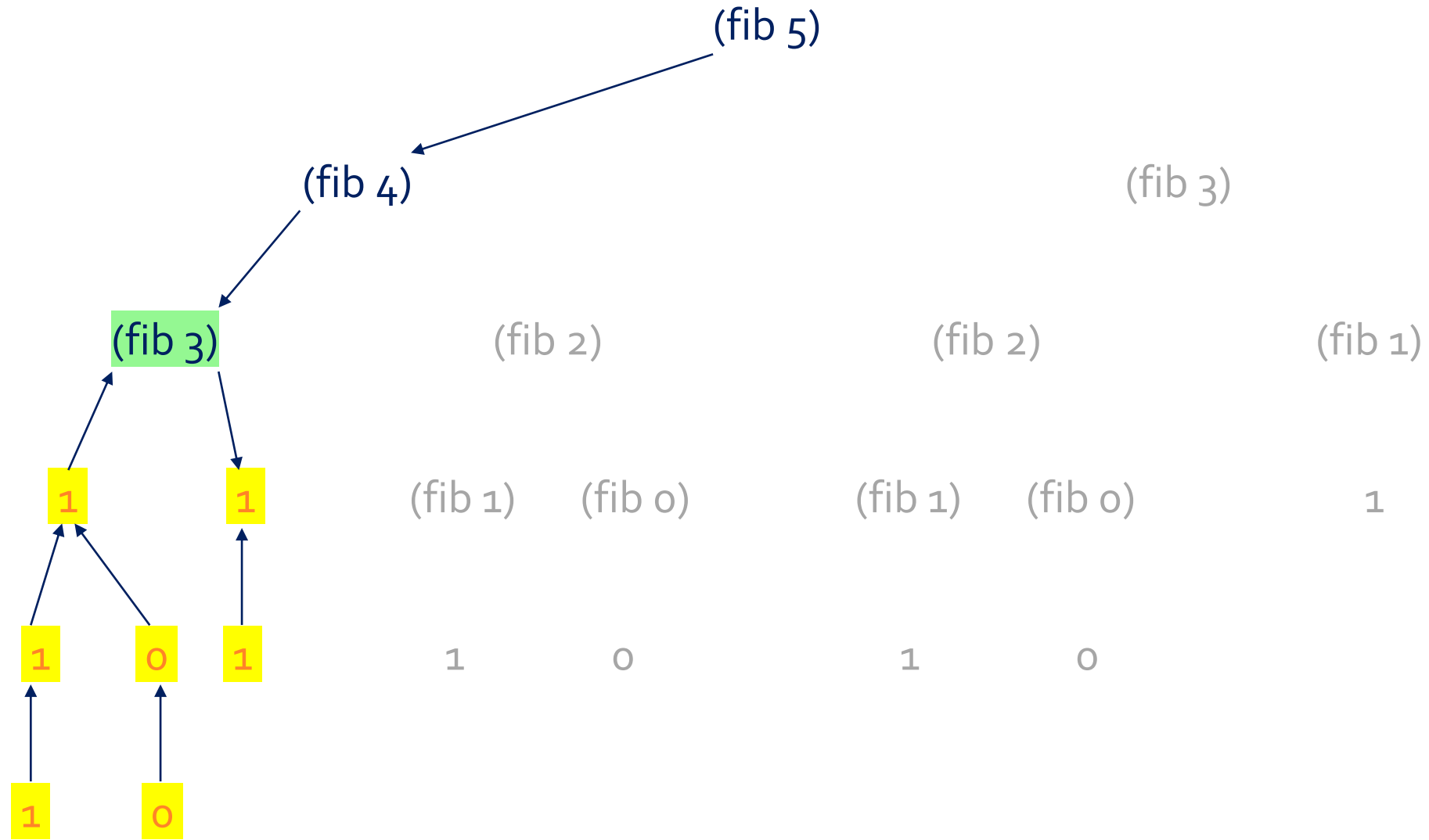


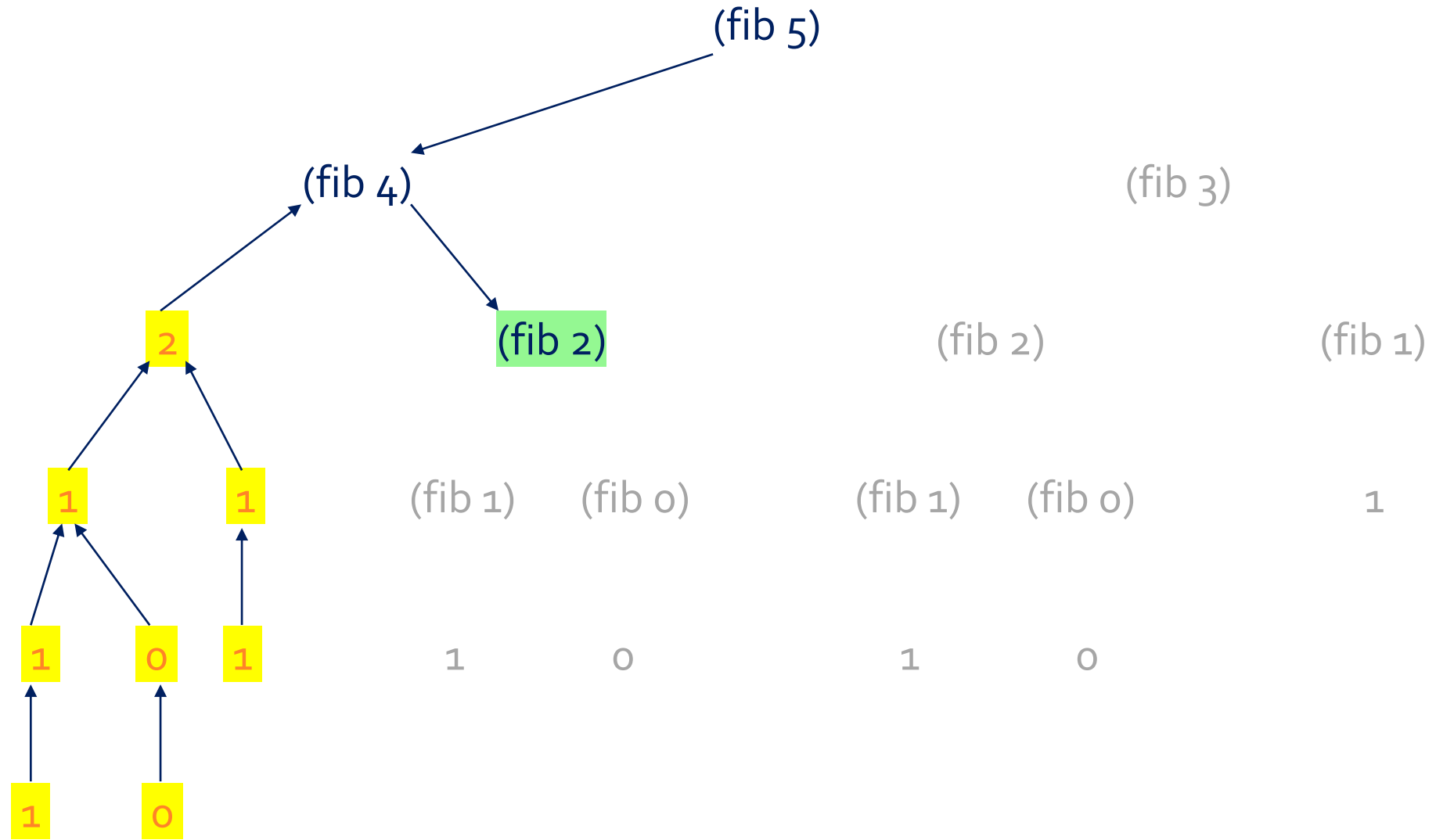


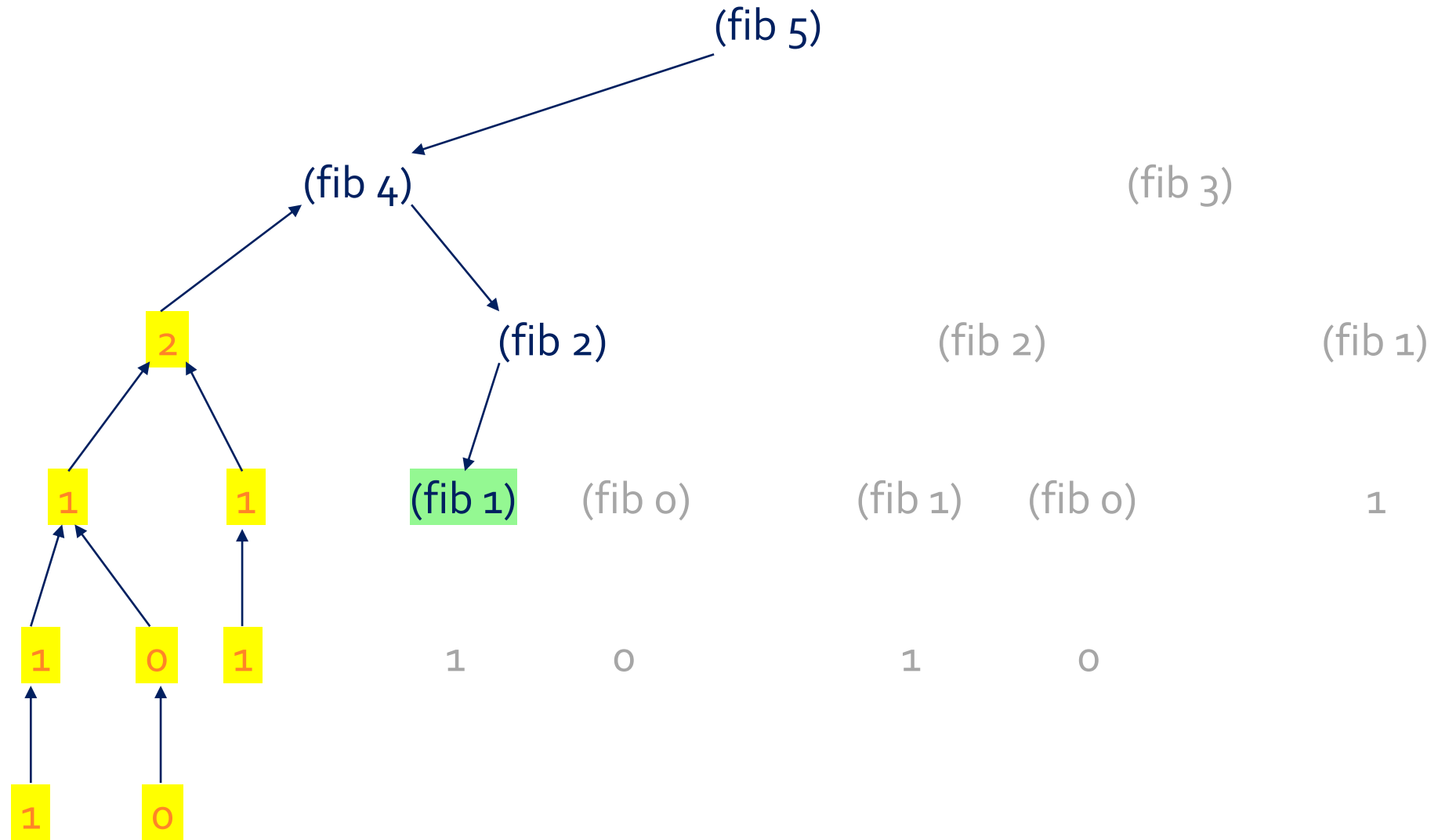




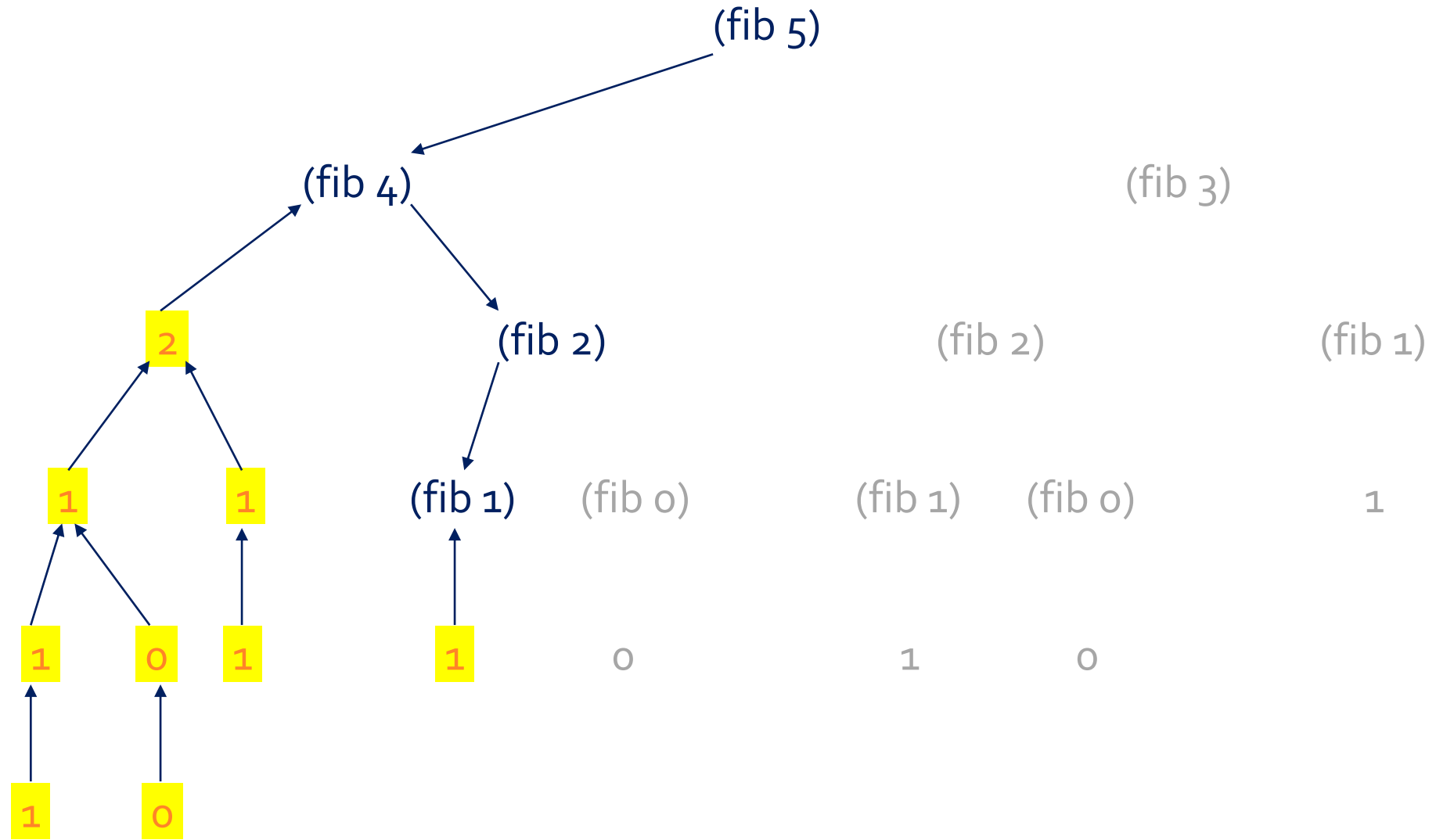


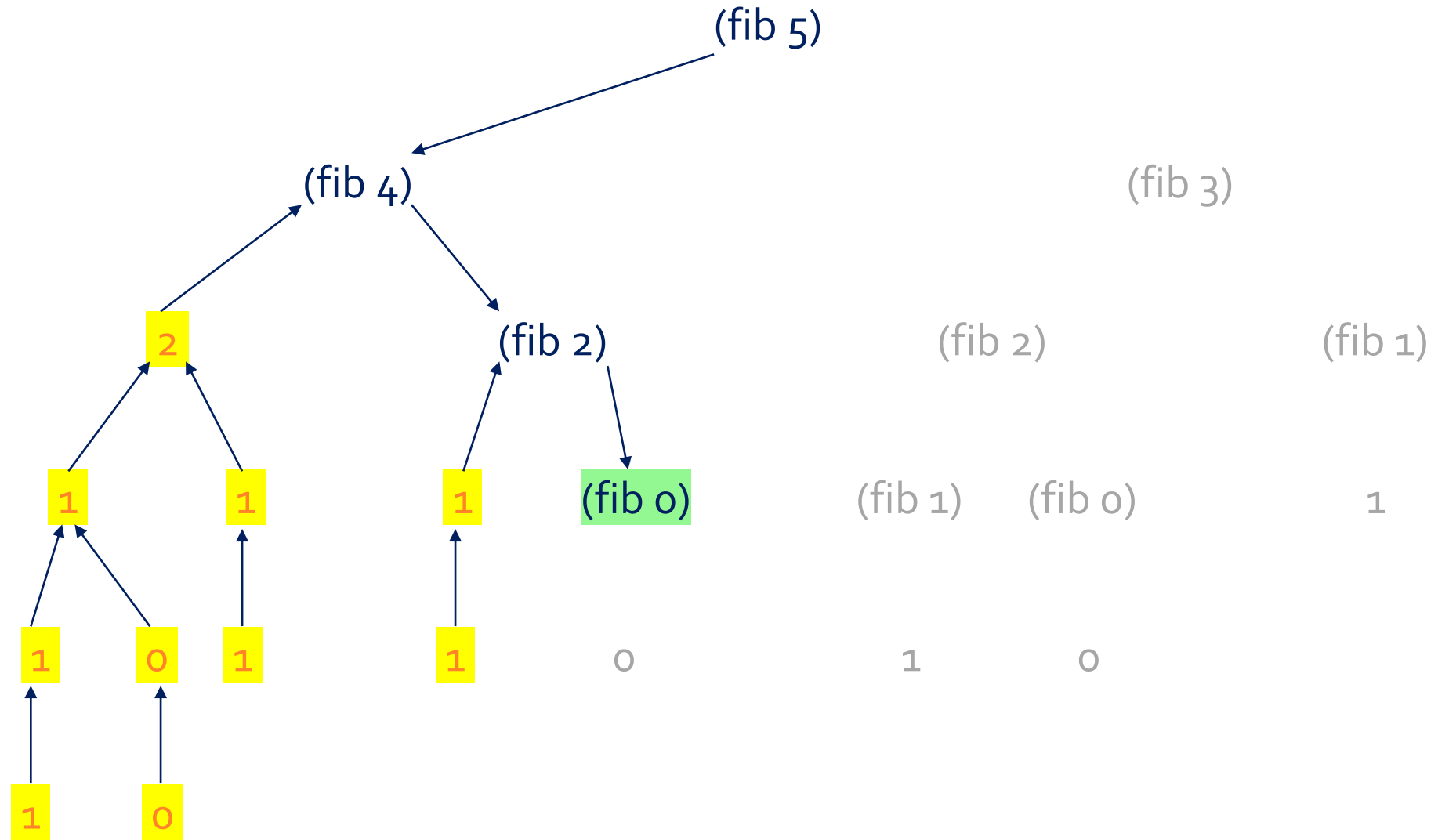


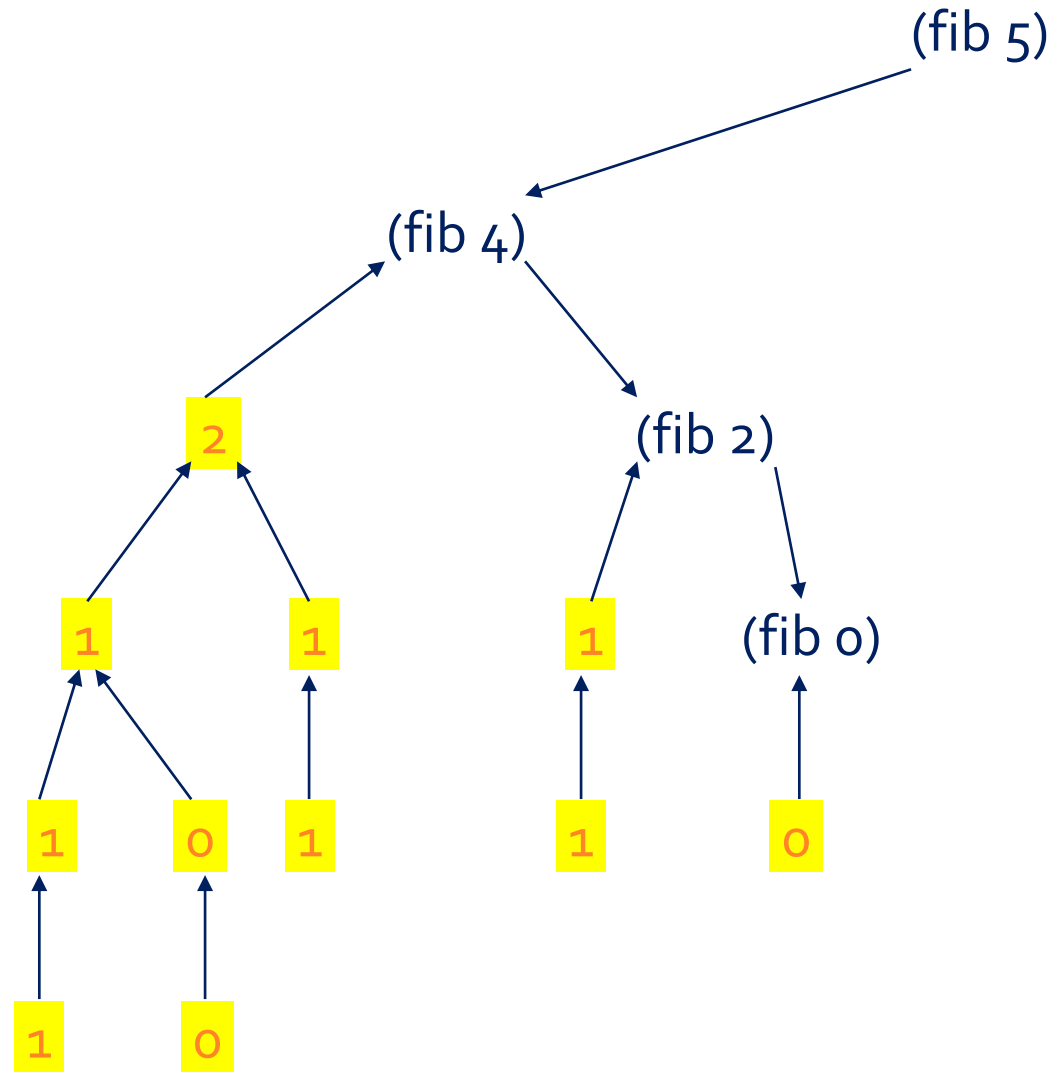


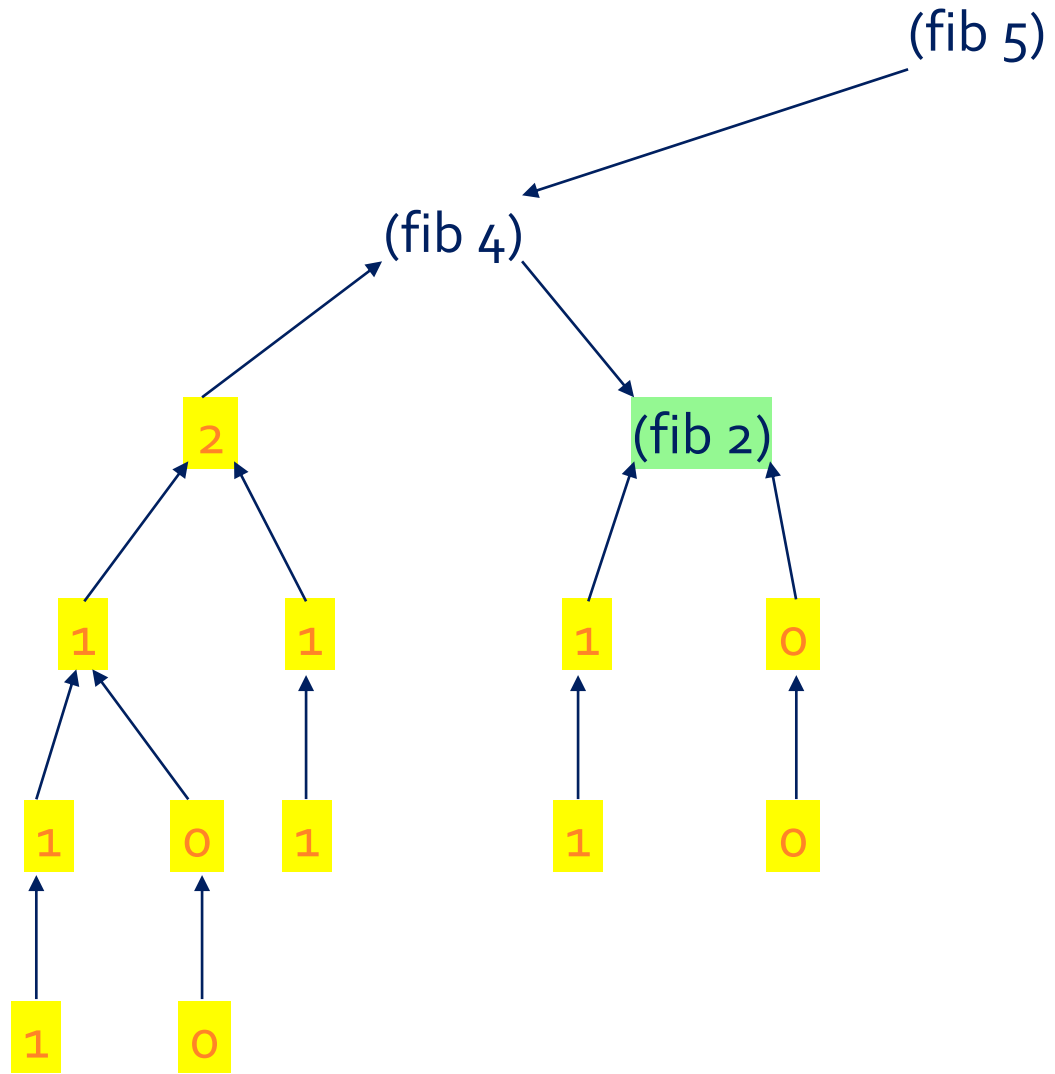


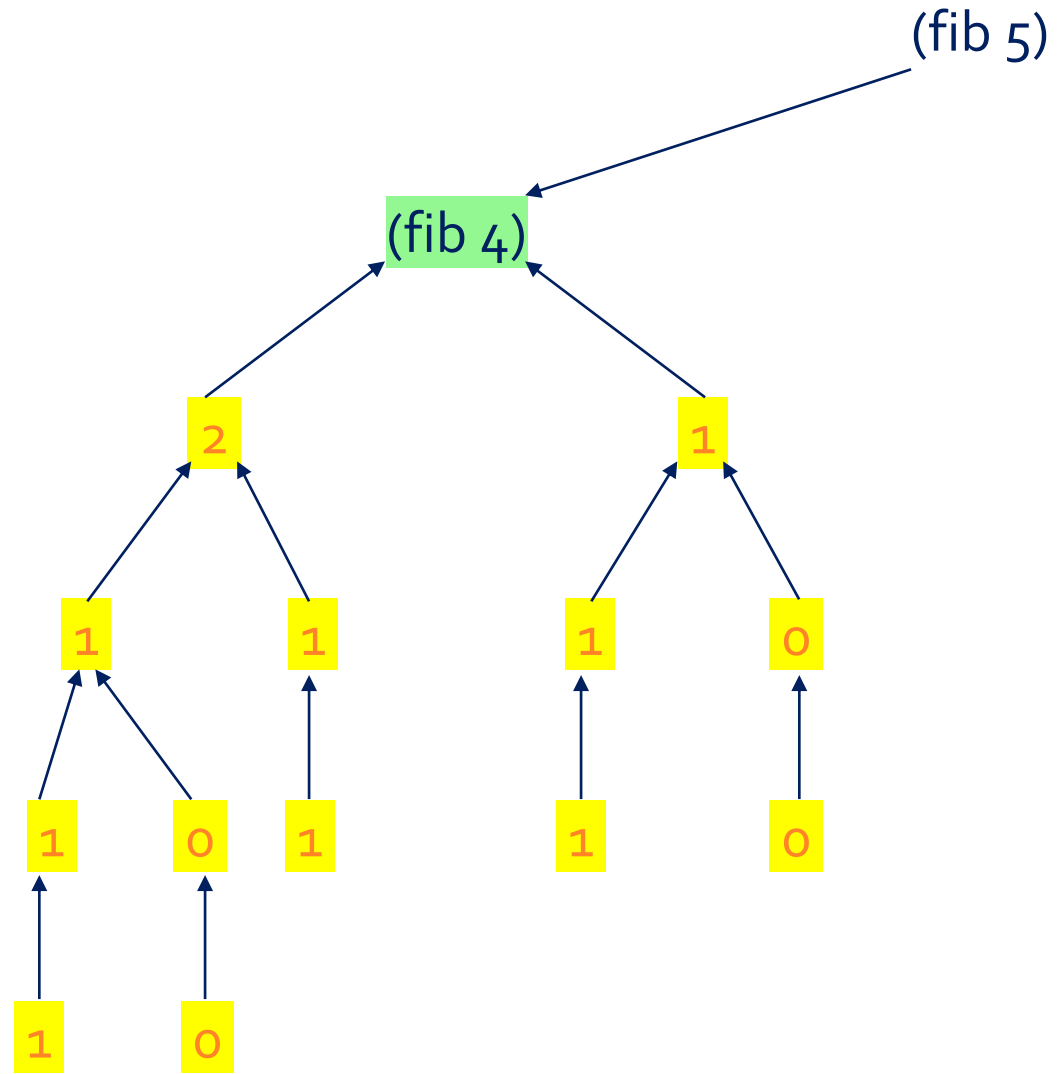


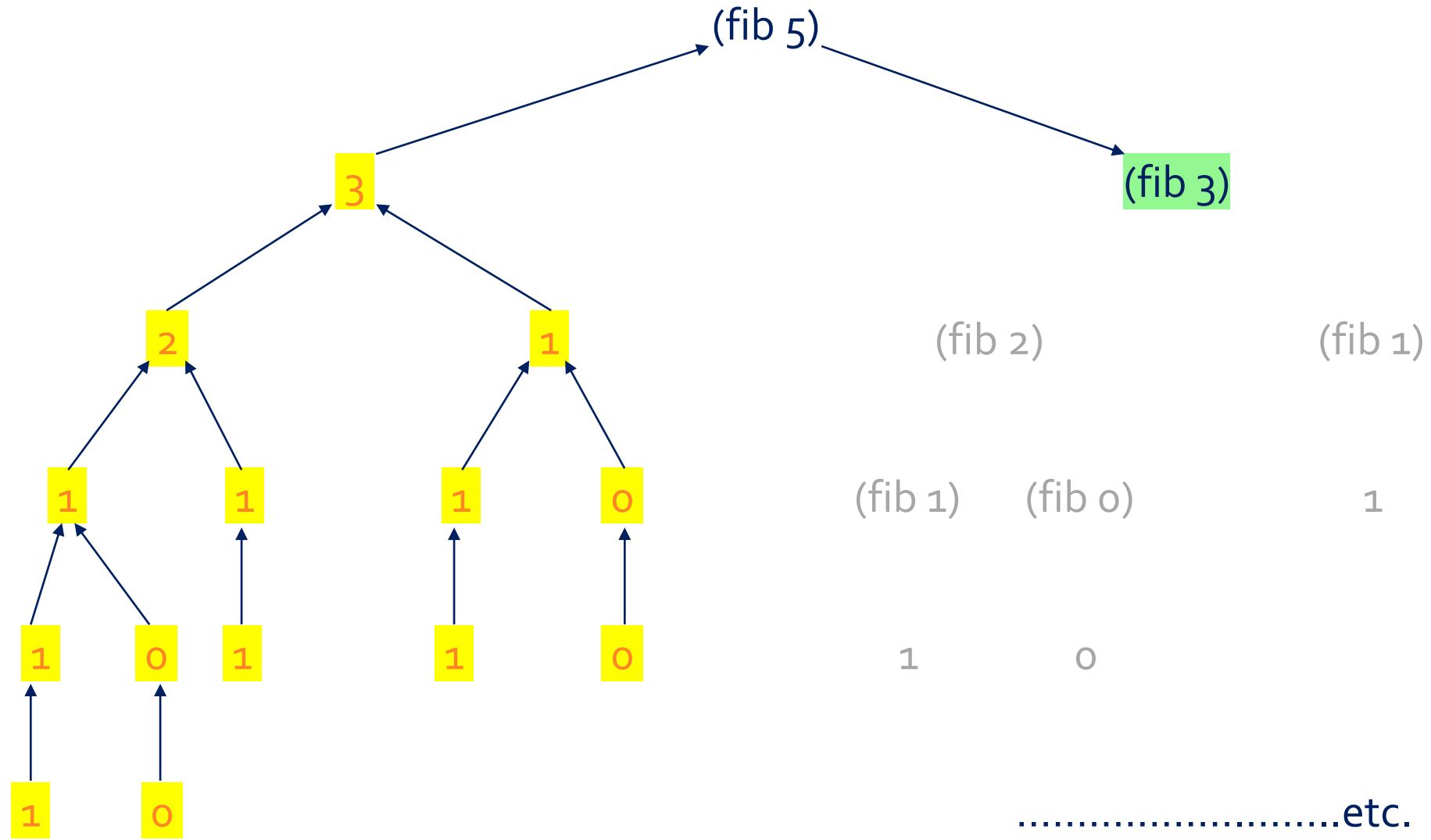








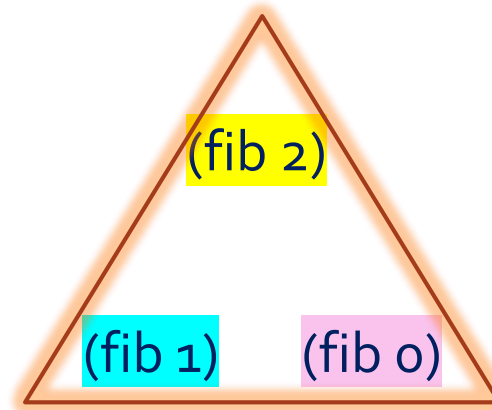
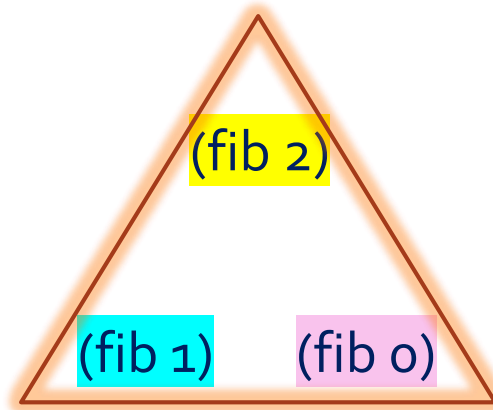
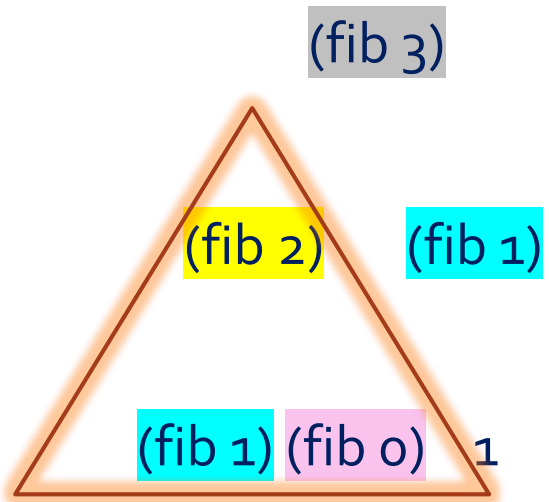




(fib 5)

(fib 4)

(fib 3)



(fib 1)

1

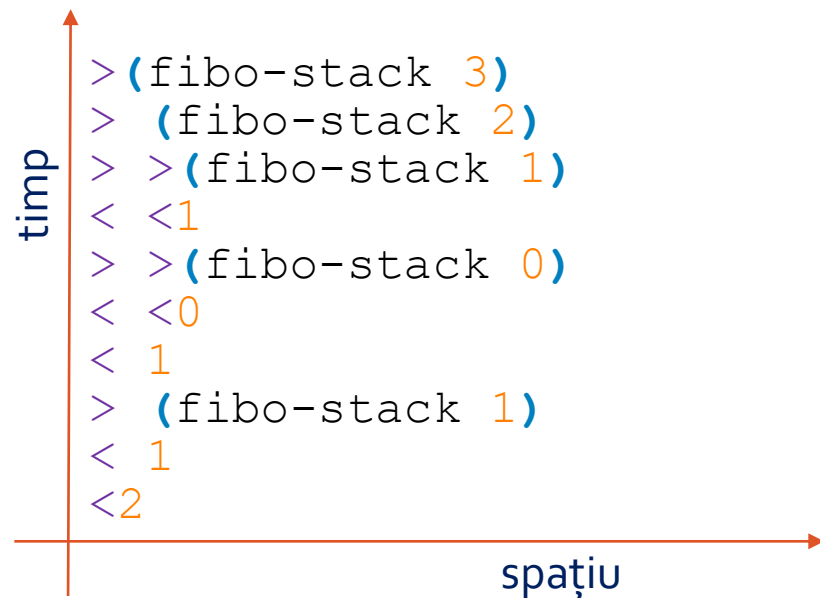
.

1 0

← multitudine de calcule duplicate

# Observații – recursivitate arborescentă

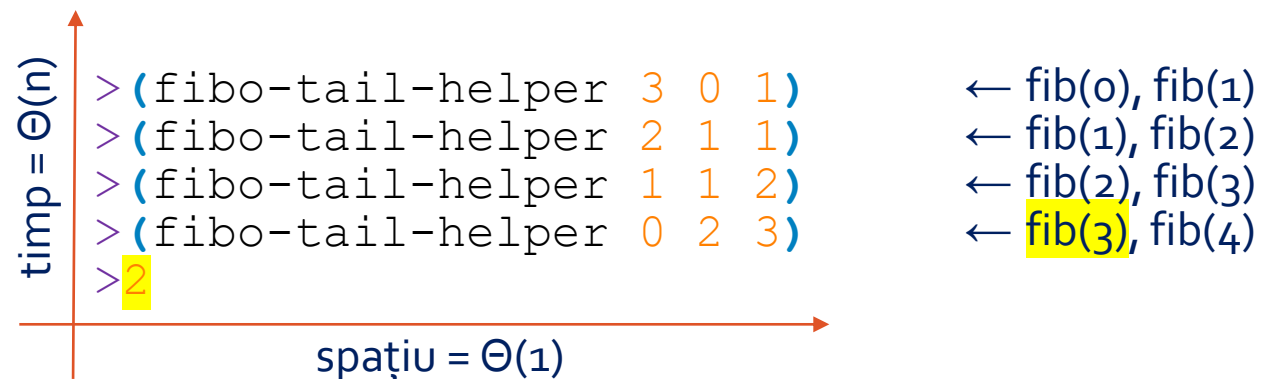
- **Timp:**  $\Theta(\text{fib}(n+1))$  (arborele are  $2 \cdot \text{fib}(n+1) - 1$  noduri)
- **Spațiu:**  $\Theta(n)$  (stiva la un moment dat reprezintă o singură cale în arbore)
- **Calcul:** realizat integral la revenirea din recursivitate





# Fibonacci cu recursivitate pe coadă

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



# 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

# 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)))))
```

# 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))))))`
- 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))))`

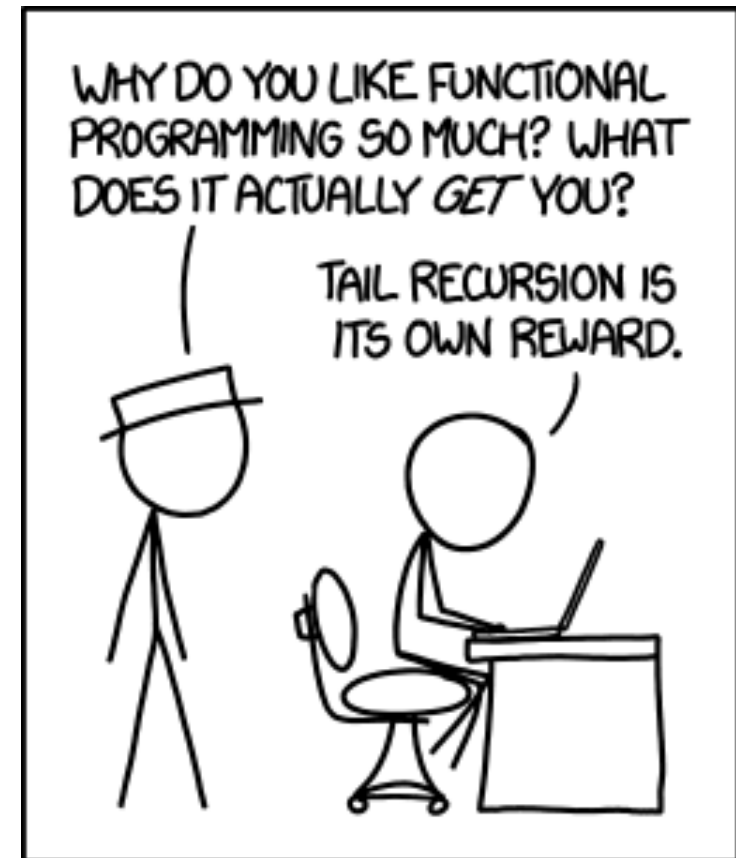
**pe stivă**

Existența unei variabile de tip  
acumulator nu înseamnă că avem  
recursivitate pe coadă!

**arborescentă**

# 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



# 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**



# 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))))))
```

```
(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-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)
```

```
>(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!

# 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:  $\Theta(n)$  (dată de complexitatea lui `reverse`)
- `append` în loc de `cons`:  $\Theta(n^2)$  ( $\Theta(\text{length}(\text{acc}))$  pentru fiecare `append` în parte)  
( $0 + 1 + 2 + \dots + (n-1)$ )

# 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)))) →  $\Theta(\text{length}(L))$ 
6.
7. (define (append A B)
8.   (if (null? A)
9.       B
10.      (cons (car A) (append (cdr A) B)))) →  $\Theta(\text{length}(A))$ 
```

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

# Calcul Lambda



Freedom  
from  
state

# Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o  $\lambda$ -expresie
- Statutul variabilelor într-o  $\lambda$ -expresie
- Evaluarea unei  $\lambda$ -expresii
- Forma normală a unei  $\lambda$ -expresii

# Memento: $\lambda$ -expresia

## Sintaxa

- $e \equiv$
- |  $x$  variabilă
  - |  $\lambda x.e1$  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  $(\lambda x.e1 e2)$  (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  $e1_{[e2/x]}$ )

# Aparițiile unei variabile într-o $\lambda$ -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 al căror nume este un alias pentru valori din exteriorul expresiei

# Apariții legate / libere într-o expresie

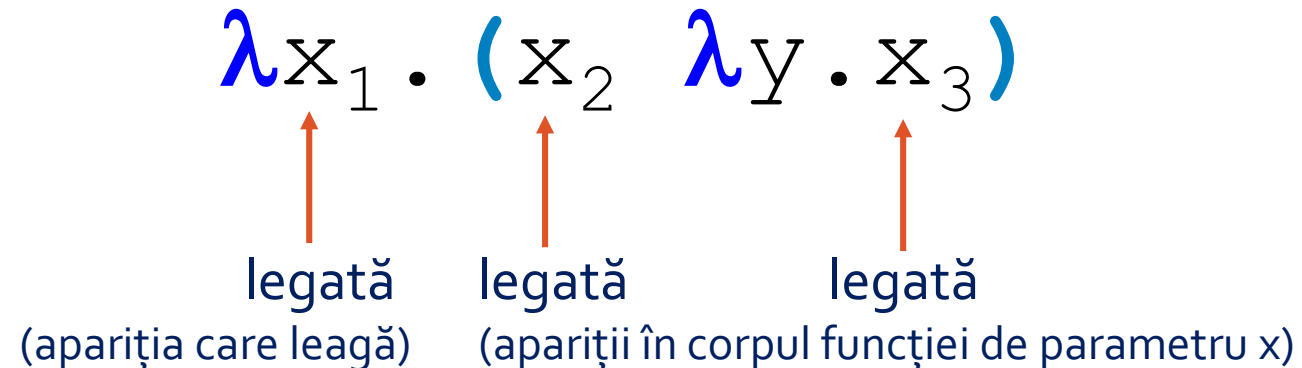
Apariția  $x_n$  este **legată** în E dacă:

- $E = \dots \lambda x_n . e \dots$
- $E = \dots \lambda x . e \dots$  și  $x_n$  apare în e

Variabila de după  $\lambda$  = variabila de legare

Apariția de după  $\lambda$  = apariția care leagă (restul aparițiilor lui x în corpul e)

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





# Exemple de apariții legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# 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)))

# Exemple de apariții legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de apariții legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de apariții legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de apariții legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de apariții legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# 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 identificador, 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)



# Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o  $\lambda$ -expresie
- Statutul variabilelor într-o  $\lambda$ -expresie
- Evaluarea unei  $\lambda$ -expresii
- Forma normală a unei  $\lambda$ -expresii

# 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:**  $(x \lambda x.x)$  din punct de vedere al statutului aparițiilor devine

$(x \lambda x.x)$  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**

# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

$(\lambda x. \lambda y.(x y) y)$  dar dacă ne limităm la subexpresia  $\lambda y.(x y)$  statutul variabilelor se inversează!

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

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

# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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



# Exemple de variabile legate / libere

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

$(x \lambda y.x)$

$(y \lambda y.x)$

$\lambda z.((+ z) x)$

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

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

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

# Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o  $\lambda$ -expresie
- Statutul variabilelor într-o  $\lambda$ -expresie
- Evaluarea unei  $\lambda$ -expresii
- Forma normală a unei  $\lambda$ -expresii

# $\beta$ -redex și $\beta$ -reducere

## Memento: Semantica $\lambda$ -expresiilor (Modelul substituției)

Pentru a evalua  $(\lambda x.e_1 e_2)$  (funcția cu parametrul formal  $x$  și corpul  $e_1$ , aplicată pe  $e_2$ ):

- Peste tot în  $e_1$ , identificatorul  $x$  este înlocuit cu  $e_2$
- Peste tot în  $e_1$ , aparițiile libere ale lui  $x$  (libere în  $e_1$  !) sunt înlocuite cu  $e_2$  (nu are rost să înlocuiesc și aparițiile legate, întrucât numele acestora este irelevant)
- Se **evaluează** noul corp  $e_1$  și se întoarce rezultatul (se notează  $e_{1[e_2/x]}$ )

**$\beta$ -redex** =  $\lambda$ -expresie de forma  $(\lambda x.e_1 e_2)$

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

# Greșeli apărute la $\beta$ -reducere

**Exemplu:**  $(\lambda x. \underbrace{\lambda y. (x \ y)}_{e1} \underbrace{y}_{e2}) \rightarrow_{\beta} \lambda y. (x \ y)_{[y/x]} = \lambda y. (y \ y)$  (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  $e2 \rightarrow$  greșeli în evaluare

# Soluția: $\alpha$ -conversia

**$\alpha$ -conversie** = redenumirea variabilelor legate din corpul unei funcții  $\lambda x.e_1$  a.î. ele să nu coincidă cu variabilele libere din parametrul efectiv  $e_2$  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  $e_1$  și din  $e_2$

**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!)

# Exemplu de secvență de reducere

$(\lambda x.(x z) \lambda z.\lambda x.(z x))$

# Exemplu de secvență de reducere

$(\lambda x.(x z) \lambda z.\lambda x.(z x)) \rightarrow_{\beta}$

$(\lambda z.\lambda x.(z x) z)$

# Exemplu de secvență de reducere

$(\lambda x.(x z) \lambda z.\lambda x.(z x)) \rightarrow_{\beta}$

$(\lambda z.\lambda x.(z x) z) \rightarrow_{\alpha}$

$(\lambda t.\lambda x.(t x) z)$



# Exemplu de secvență de reducere

$(\lambda x.(x z) \lambda z.\lambda x.(z x)) \rightarrow_{\beta}$

$(\lambda z.\lambda x.(z x) z) \rightarrow_{\alpha}$

$(\lambda t.\lambda x.(t x) z) \rightarrow_{\beta}$

$\lambda x.(z x)$

# Calcul Lambda – Cuprins

- Aparițiile unei variabile într-o  $\lambda$ -expresie
- Statutul variabilelor într-o  $\lambda$ -expresie
- Evaluarea unei  $\lambda$ -expresii
- Forma normală a unei  $\lambda$ -expresii

# Forma normală a unei $\lambda$ -expresii

$\lambda$ -expresie în **forma normală**  $\Leftrightarrow$   $\lambda$ -expresie care nu conține niciun  $\beta$ -redex

## Întrebări

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

**Observație** (ajutătoare pentru întrebările anterioare)

Calculul Lambda este un model de calculabilitate.

$\lambda$ -expresiile sunt practic programe capabile să ruleze pe o ipotetică Mașină Lambda.

# Are orice $\lambda$ -expresie o formă normală?

**NU.**

**Exemplu:**  $(\lambda x.(x x) \lambda x.(x x)) \rightarrow_{\beta} (\lambda x.(x x) \lambda x.(x x)) \rightarrow_{\beta} (\lambda x.(x x) \lambda x.(x x)) \rightarrow_{\beta} \dots$

$\lambda$ -expresie **reductibilă**  $\Leftrightarrow$  admite o secvență finită de reducere până la o formă normală

Altfel,  $\lambda$ -expresia este **ireductibilă**.

# Forma normală este unică?

DA.

## Teorema Church-Rosser

Dacă  $\left\{ \begin{array}{l} e \rightarrow^* a \\ \text{și} \\ e \rightarrow^* b \end{array} \right.$  atunci  $\exists d$  a.î.  $\left\{ \begin{array}{l} a \rightarrow^* d \\ \text{și} \\ b \rightarrow^* d \end{array} \right.$  ( $\rightarrow^*$  = secvență de reducere)

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

# Se poate garanta găsirea forme normale?

DA.

## Teorema normalizării

Pentru orice  $\lambda$ -expresie reductibilă, se poate ajunge la forma ei normală aplicând **reducere stânga- $\rightarrow$ dreapta** (reducând mereu cel mai din stânga  $\beta$ -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- $\rightarrow$ stânga nu s-ar termina niciodată

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

# Rezumat

Teza lui Church

Tipuri de recursivitate

Apariții ale unei variabile într-o expresie

Variabile într-o expresie

$\beta$ -redex și  $\beta$ -reducere

$\alpha$ -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

# 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

$\beta$ -redex și  $\beta$ -reducere

$\alpha$ -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării



# 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

$\beta$ -redex și  $\beta$ -reducere

$\alpha$ -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

# 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 ( $\lambda \underline{x}.e$ ,  $\lambda x. \dots \underline{x} \dots$ ), libere (restul)

Variabile într-o expresie

$\beta$ -redex și  $\beta$ -reducere

$\alpha$ -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

# 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 ( $\lambda x.e$ ,  $\lambda x. \dots x \dots$ ), libere (restul)

**Variabile într-o expresie:** legate (toate aparițiile sunt legate), libere (restul)

$\beta$ -redex și  $\beta$ -reducere

$\alpha$ -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

# 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 ( $\lambda \underline{x}.e$ ,  $\lambda x. \dots \underline{x} \dots$ ), libere (restul)

**Variabile într-o expresie:** legate (toate aparițiile sunt legate), libere (restul)

**$\beta$ -redex și  $\beta$ -reducere:**  $(\lambda x.e_1 e_2), (\lambda x.e_1 e_2) \rightarrow_{\beta} e_1_{[e_2/x]}$

$\alpha$ -conversie

Forma normală

Expresie ireductibilă

Teorema normalizării

# 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 ( $\lambda \underline{x}.e$ ,  $\lambda x. \dots \underline{x} \dots$ ), libere (restul)

**Variabile într-o expresie:** legate (toate aparițiile sunt legate), libere (restul)

**$\beta$ -redex și  $\beta$ -reducere:**  $(\lambda x.e_1 e_2), (\lambda x.e_1 e_2) \rightarrow_{\beta} e_1_{[e_2/x]}$

**$\alpha$ -conversie:**  $(\lambda x. \dots \underline{\lambda y}.e_1 \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda t}.e_1_{[t/y]} \dots e_2)$  unde  $t$  nu era liberă în  $e_1, e_2$

Forma normală

Expresie ireductibilă

Teorema normalizării

# 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 ( $\lambda \underline{x}.e$ ,  $\lambda x. \dots \underline{x} \dots$ ), libere (restul)

**Variabile într-o expresie:** legate (toate aparițiile sunt legate), libere (restul)

**$\beta$ -redex și  $\beta$ -reducere:**  $(\lambda x.e_1 e_2), (\lambda x.e_1 e_2) \rightarrow_{\beta} e_1_{[e_2/x]}$

**$\alpha$ -conversie:**  $(\lambda x. \dots \underline{\lambda y}.e_1 \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda t}.e_1_{[t/y]} \dots e_2)$  unde  $t$  nu era liberă în  $e_1, e_2$

**Forma normală:**  $\lambda$ -expresia nu conține niciun  $\beta$ -redex

Expresie ireductibilă

Teorema normalizării

# 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 ( $\lambda \underline{x}.e$ ,  $\lambda x. \dots \underline{x} \dots$ ), libere (restul)

**Variabile într-o expresie:** legate (toate aparițiile sunt legate), libere (restul)

**$\beta$ -redex și  $\beta$ -reducere:**  $(\lambda x. e_1 e_2), (\lambda x. e_1 e_2) \rightarrow_{\beta} e_{1[e_2/x]}$

**$\alpha$ -conversie:**  $(\lambda x. \dots \underline{\lambda y}. e_1 \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda t}. e_{1[t/y]} \dots e_2)$  unde  $t$  nu era liberă în  $e_1, e_2$

**Forma normală:**  $\lambda$ -expresia nu conține niciun  $\beta$ -redex

**Expresie ireductibilă:** nu poate fi redusă la o formă normală (altfel – reductibilă)

Teorema normalizării

# 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 ( $\lambda \underline{x}.e$ ,  $\lambda x. \dots \underline{x} \dots$ ), libere (restul)

**Variabile într-o expresie:** legate (toate aparițiile sunt legate), libere (restul)

**$\beta$ -redex și  $\beta$ -reducere:**  $(\lambda x.e_1 e_2), (\lambda x.e_1 e_2) \rightarrow_{\beta} e_{1[e_2/x]}$

**$\alpha$ -conversie:**  $(\lambda x. \dots \underline{\lambda y}.e_1 \dots e_2) \rightarrow_{\alpha} (\lambda x. \dots \underline{\lambda z}.e_{1[z/y]} \dots e_2)$  unde  $z$  nu era liberă în  $e_1, e_2$

**Forma normală:**  $\lambda$ -expresia nu conține niciun  $\beta$ -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)