# PARADIGME DE PROGRAMARE

## Curs 4

Recursivitate pe stivă / 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

# 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

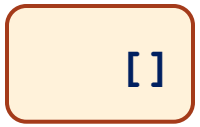
# Exemplu – recursivitate pe stivă

```
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

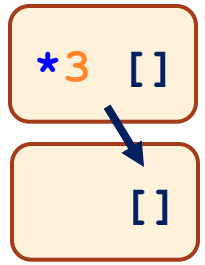



Stiva procesului

# 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))
```

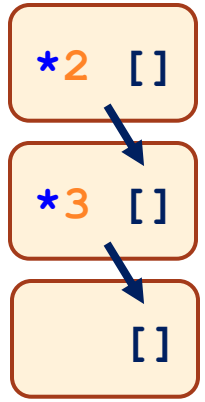



Stiva procesului

# 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)))
```

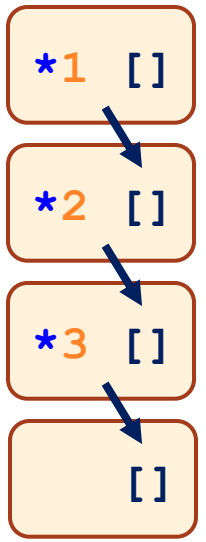



Stiva procesului

# 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))))
```

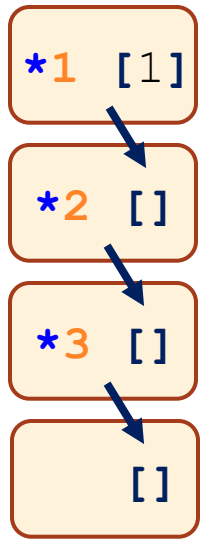



Stiva procesului

# 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)))
```

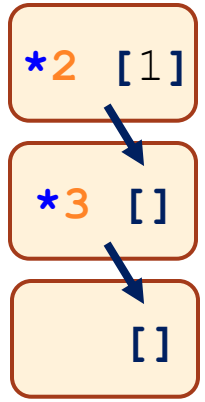



Stiva procesului

# 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))
```

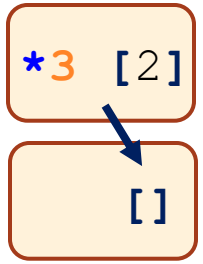



Stiva procesului

# 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)))))
```

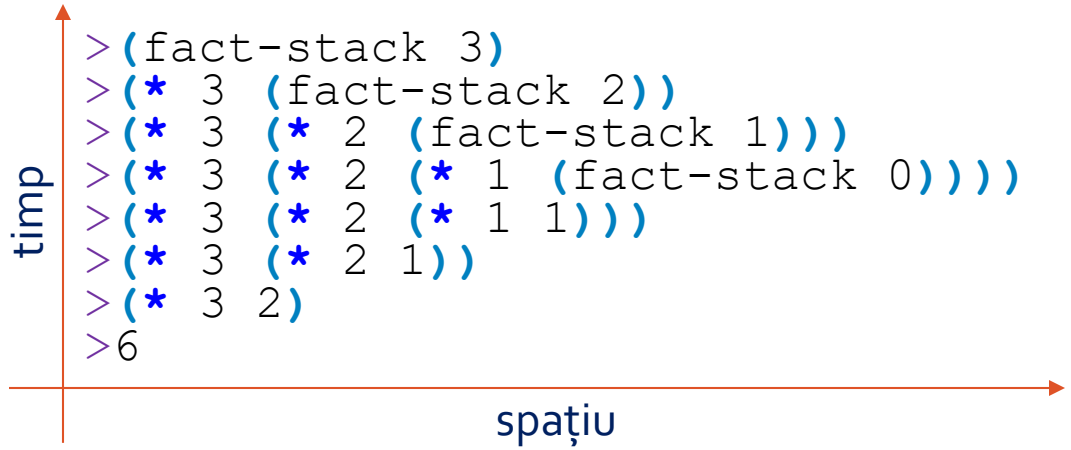
```
>(fact-stack 3)
>(* 3 (fact-stack 2))
>(* 3 (* 2 (fact-stack 1)))
>(* 3 (* 2 (* 1 (fact-stack 0))))
>(* 3 (* 2 (* 1 1)))
>(* 3 (* 2 1))
>(* 3 2)
>6
```

[6]

Stiva procesului

# Observații – recursivitate pe stivă

- Timp: 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)



# 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?

# 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

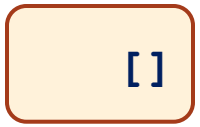
# Exemplu – recursivitate pe coadă

```
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)
```

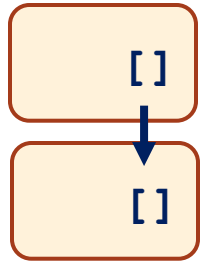



Stiva procesului

# 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)
```

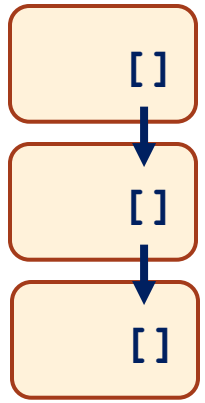



Stiva procesului

# 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)
```

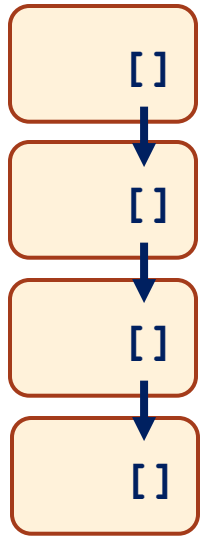



Stiva procesului

# 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)
```



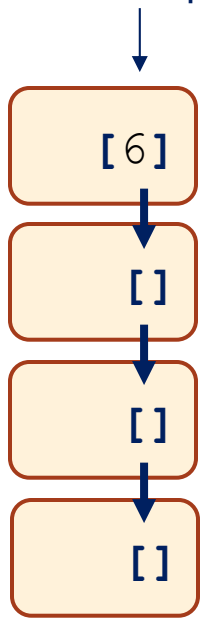
Stiva procesului

# 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

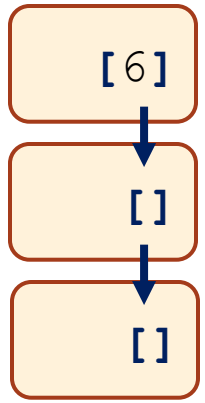



Stiva procesului

# 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
```

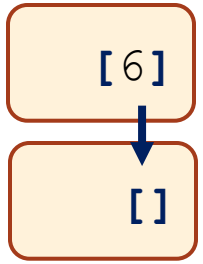



Stiva procesului

# 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
```

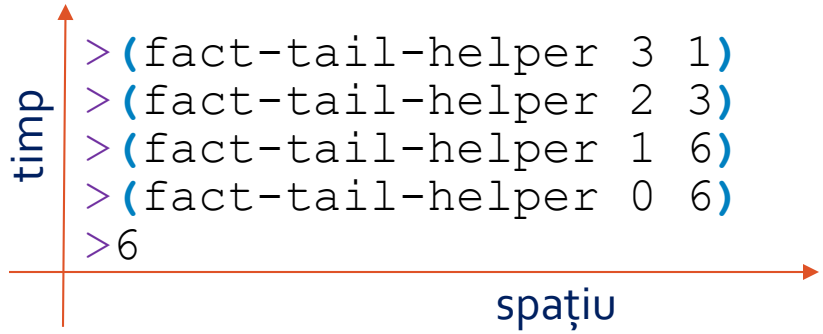
[6]

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: Θ(n) (se efectuează n înmulțiri)
- Spațiu: Θ(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

# 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

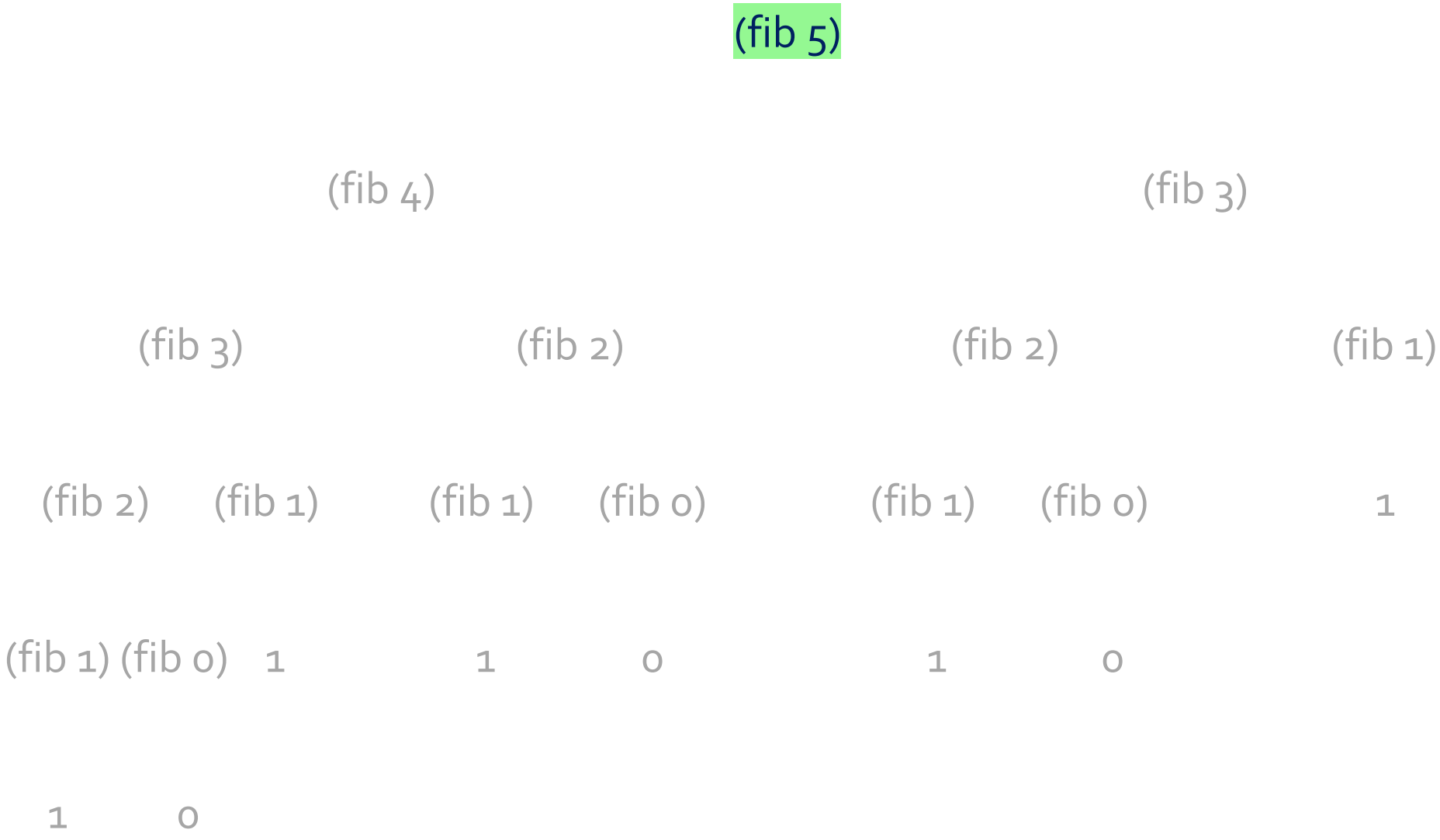
# Exemplu – recursivitate arborescentă

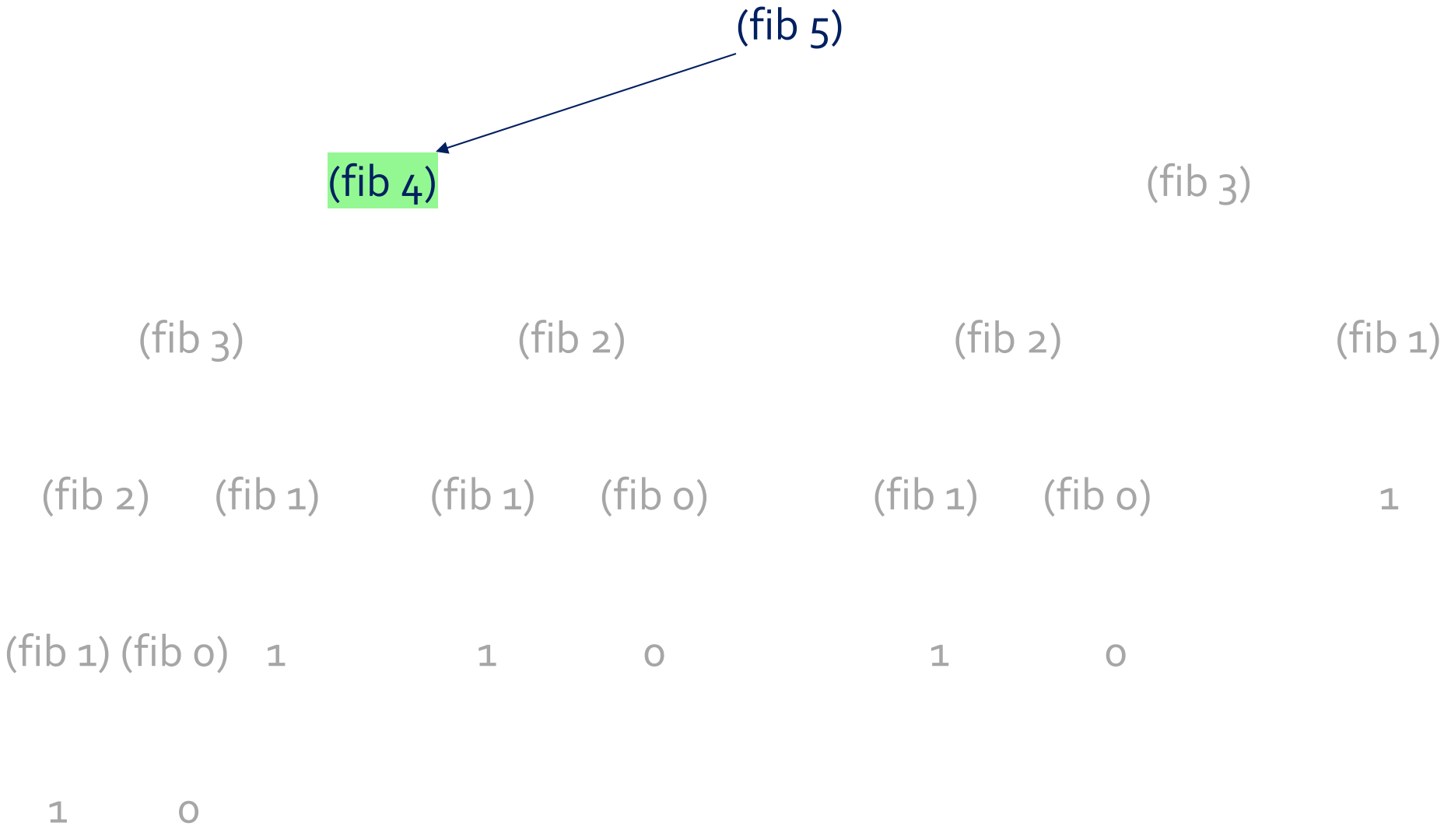
```
1.  (define (fibo-stack n)
2.    (if (< n 2)
3.        n
4.        (+ (fibo-stack (- n 1)) (fibo-stack (- n 2)))))
```

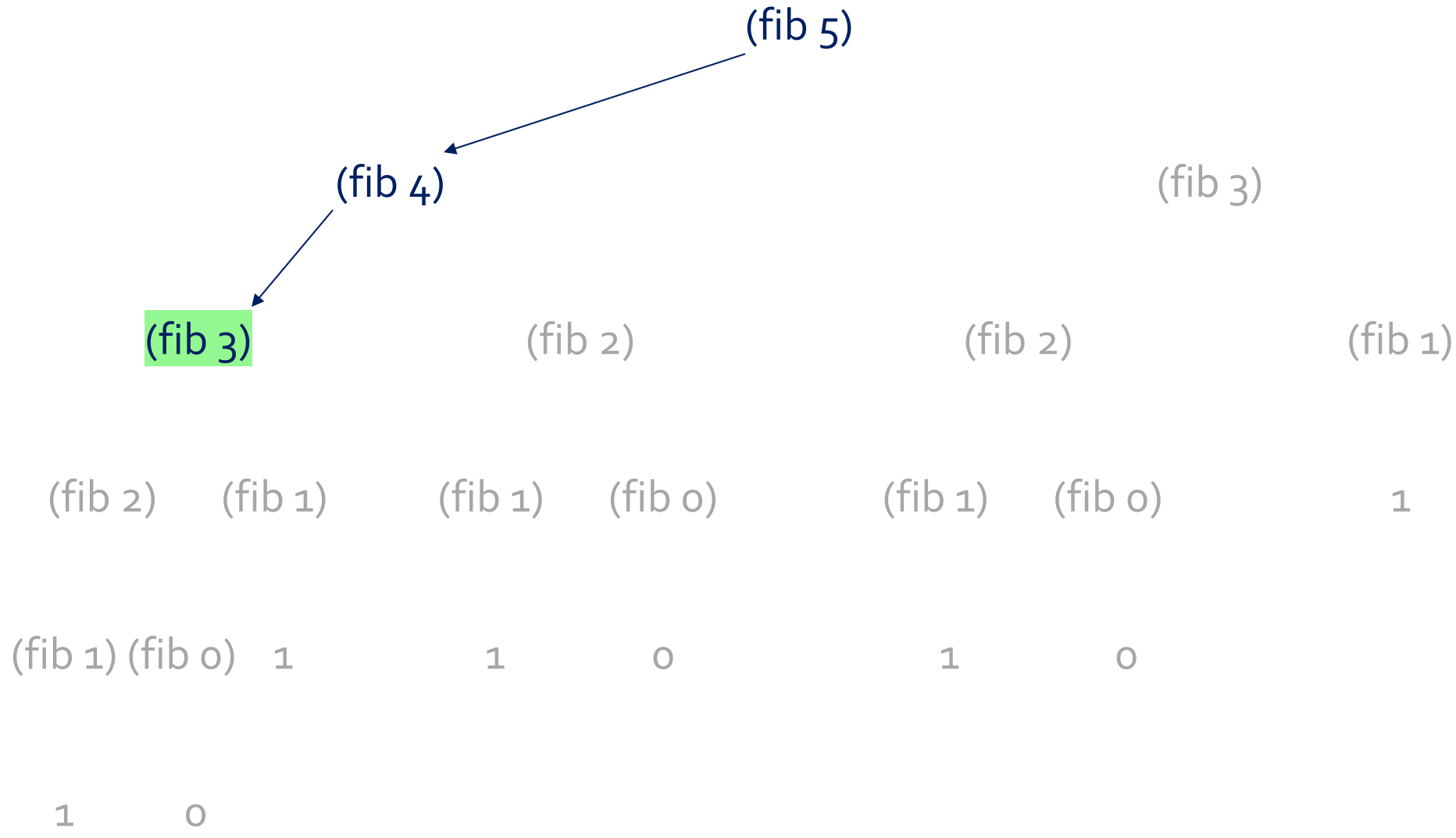
rezultatele a 2 apeluri recursive sunt așteptate pentru a fi adunate între ele

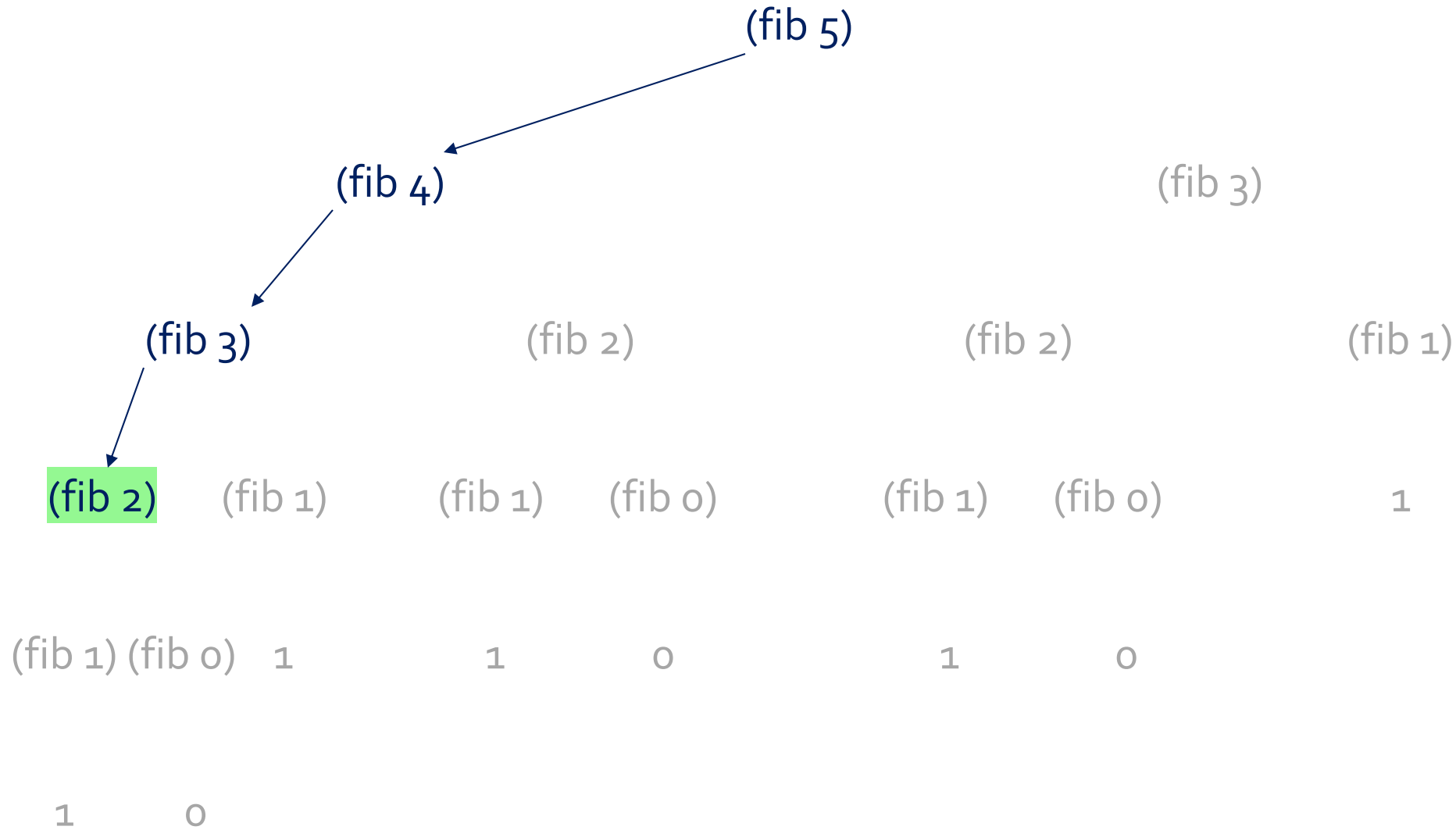
```
>(fibo-stack 3)
> (fibo-stack 2)
> >(fibo-stack 1)
< <1
> >(fibo-stack 0)
< <0
< 1
> (fibo-stack 1)
< 1
<2
```

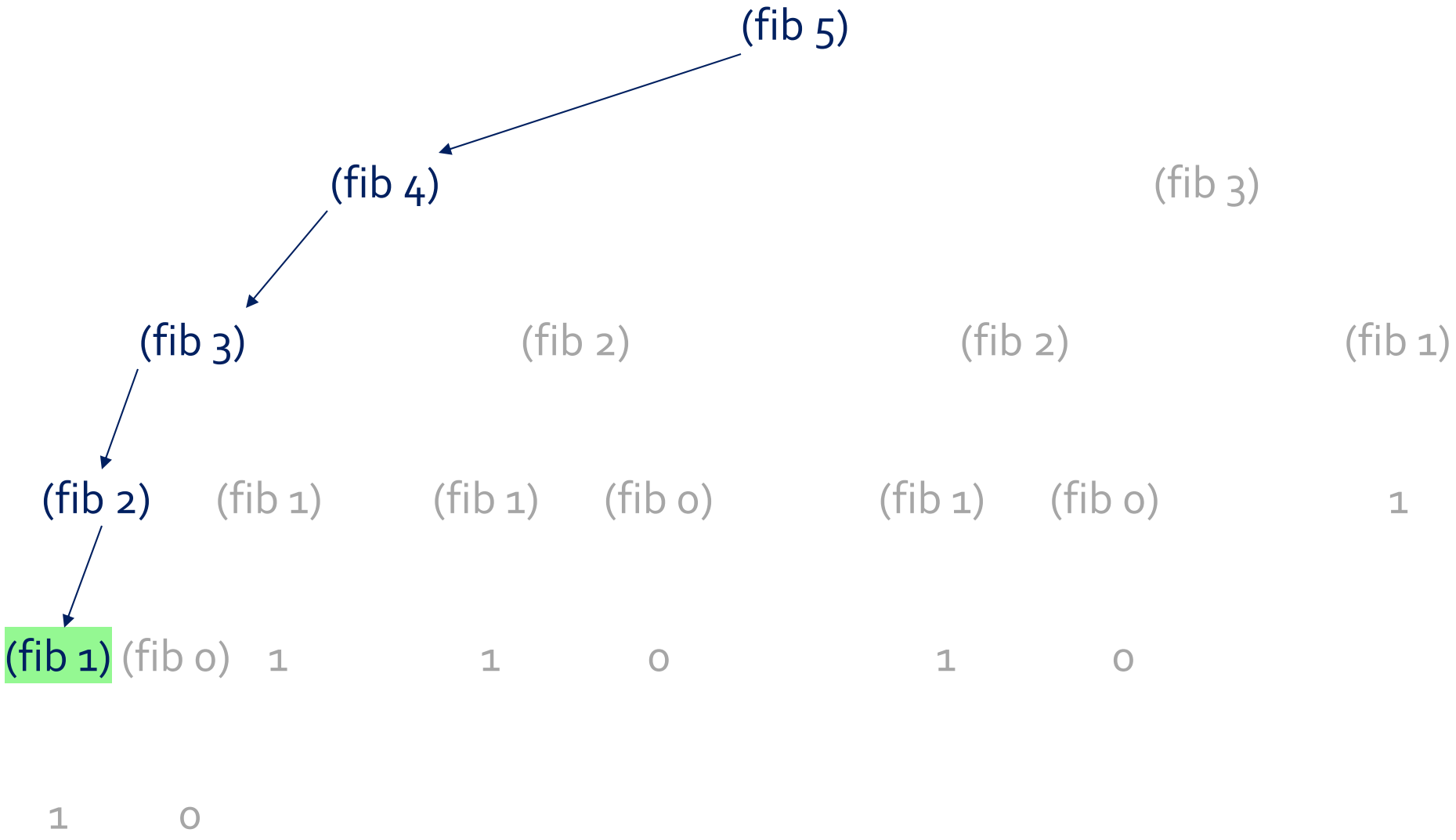
← `(fibo-stack 1)` se calculează de 2 ori, iar numărul de calcule redundante crește exponențial cu n

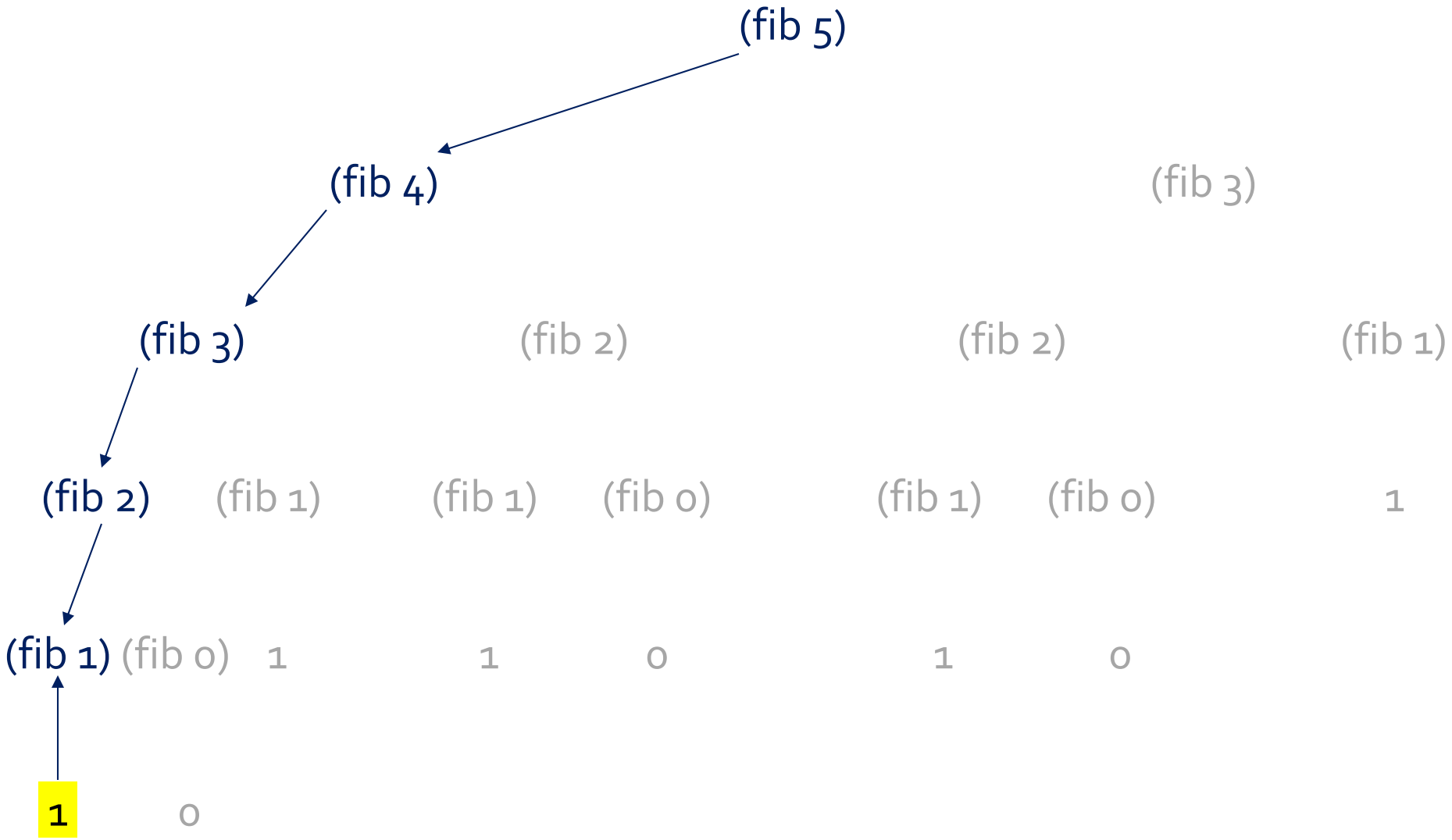

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
(fib 1)
(fib 0)
1
1
0
1
0
1
0

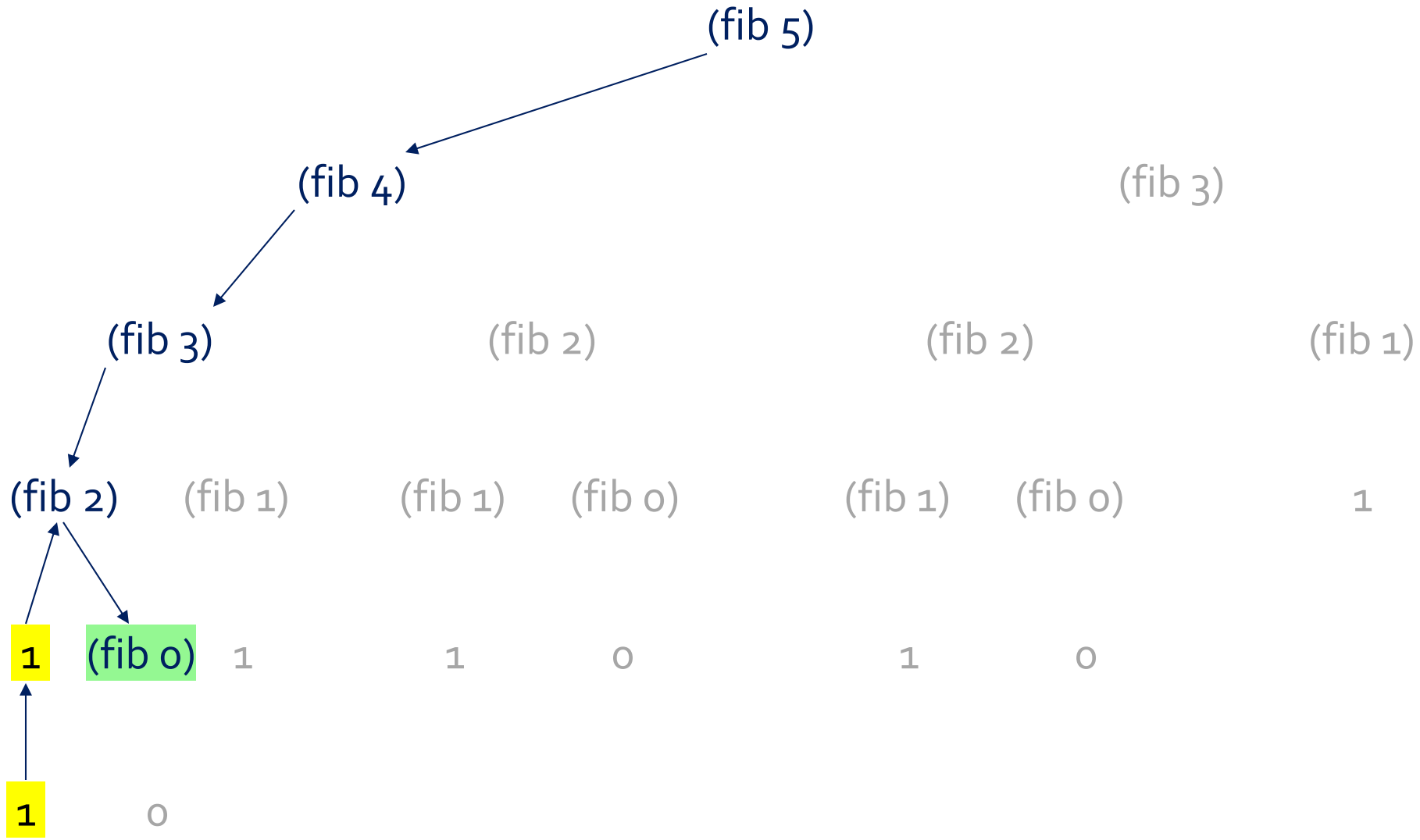

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
1
0
1
0
1
0
1

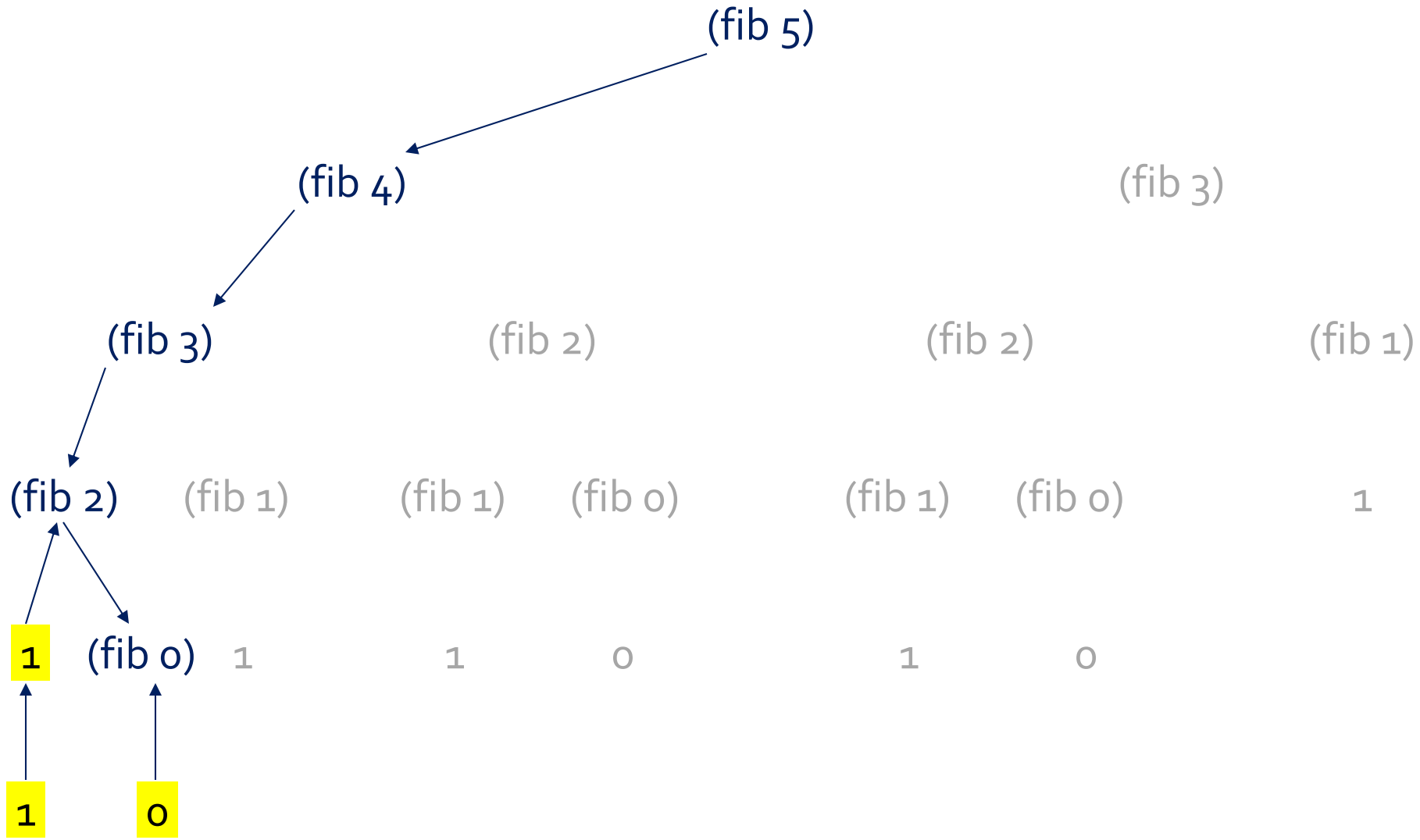

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
(fib 1)
(fib 0)
1
1
0
1
0
1
0

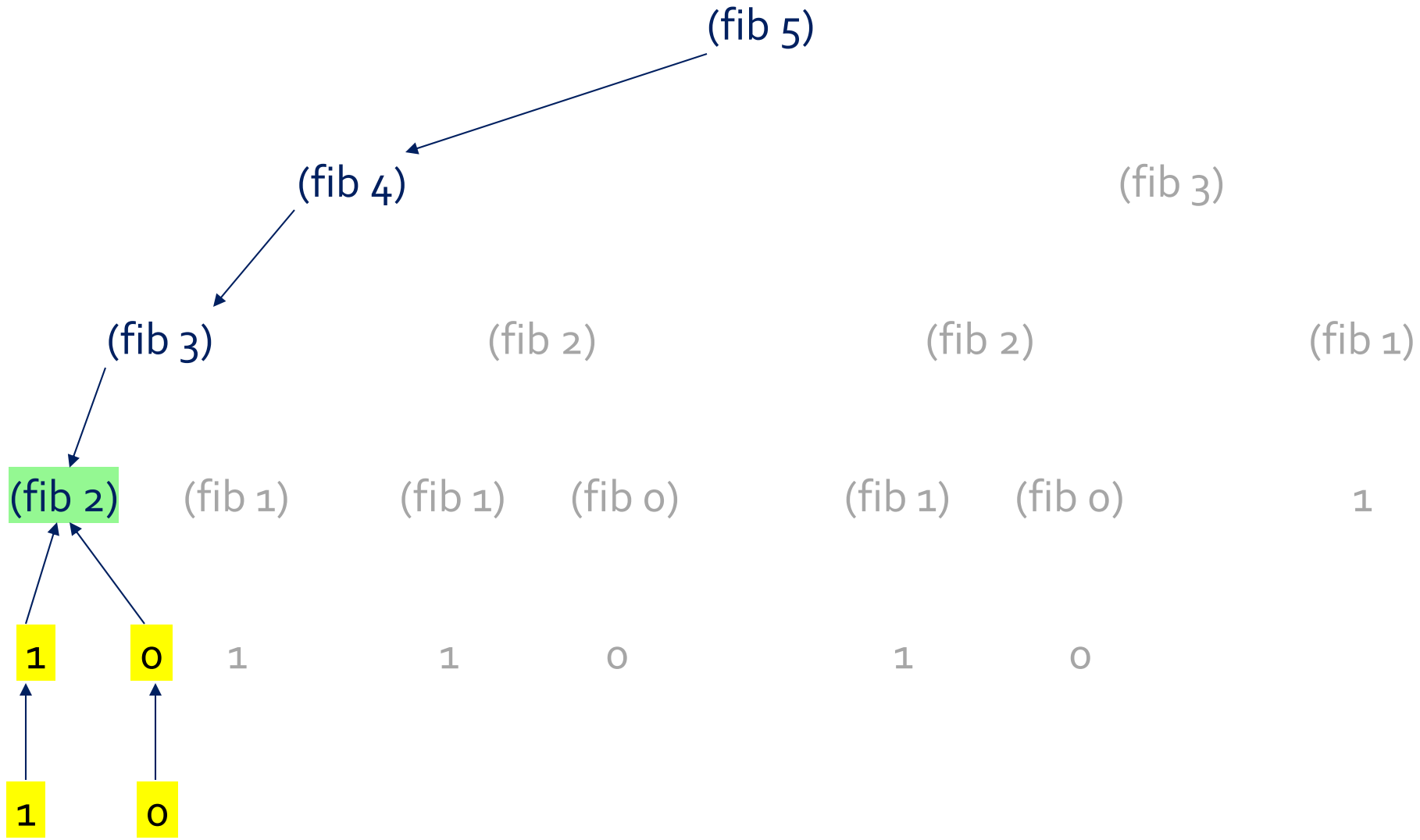

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
(fib 1)
(fib 0)
1
1
0
1
0
1
0

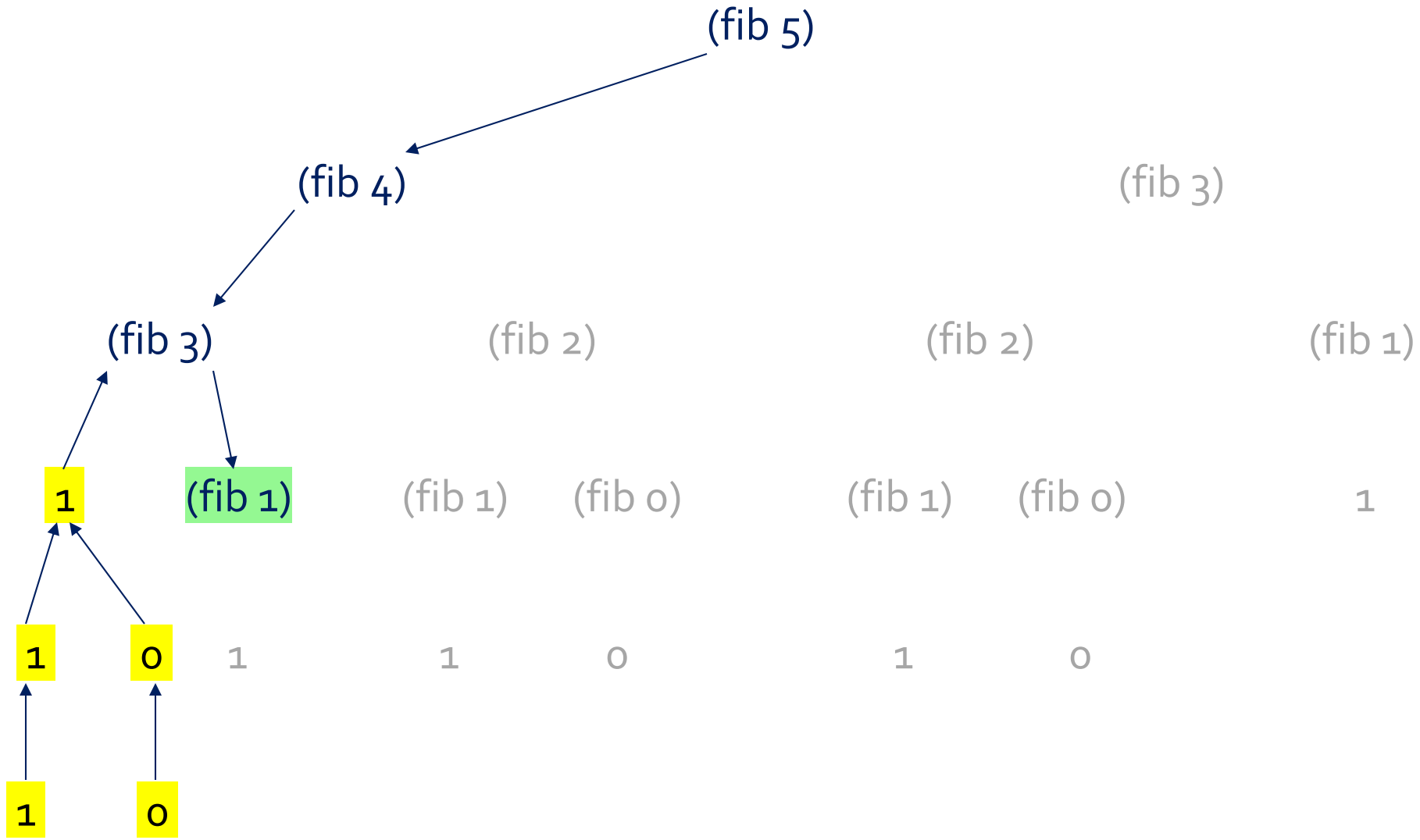

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
(fib 1)
(fib 0)
1
1
0
1
0
1
0

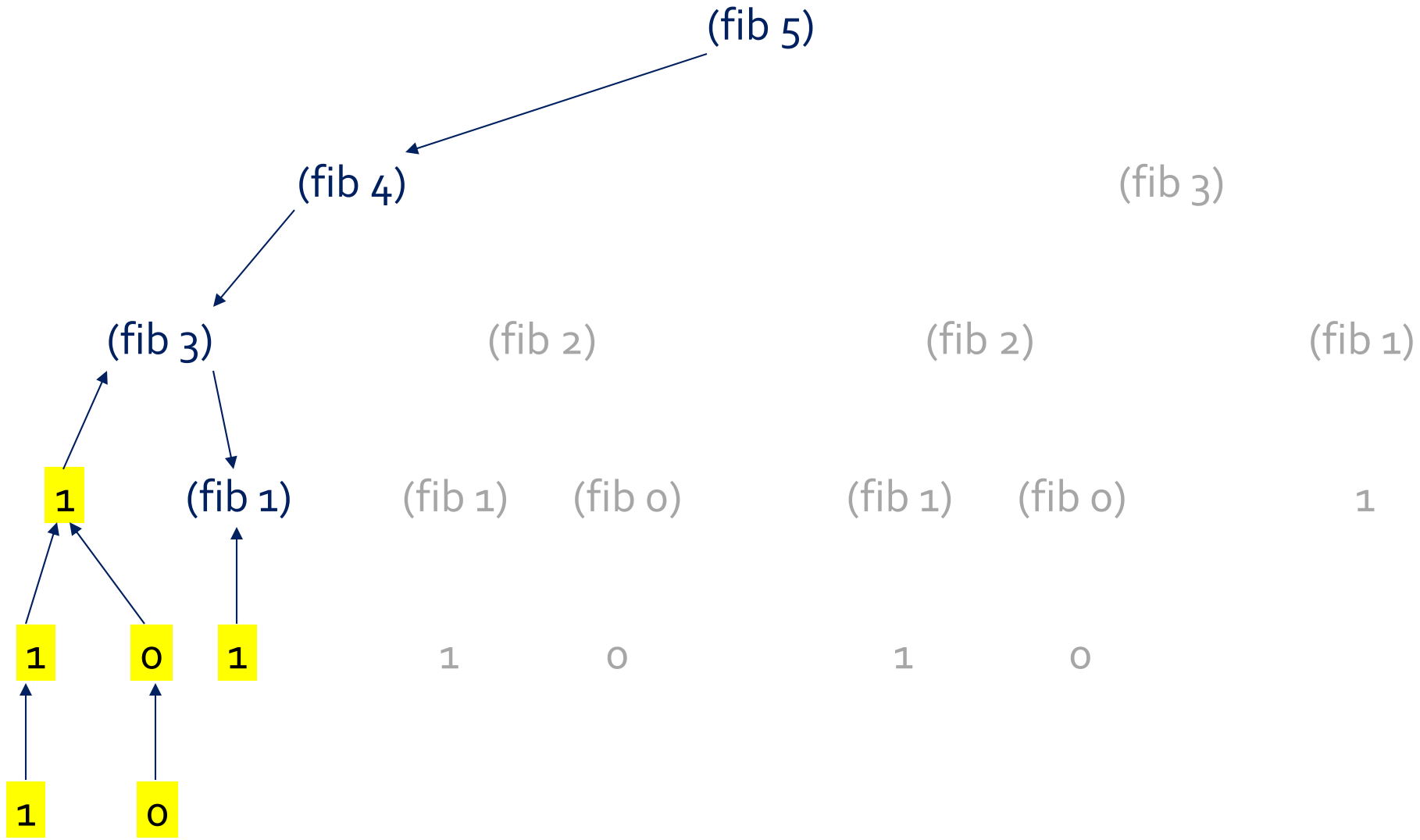

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
(fib 1)
(fib 0)
1
1
0
1
0
1
0

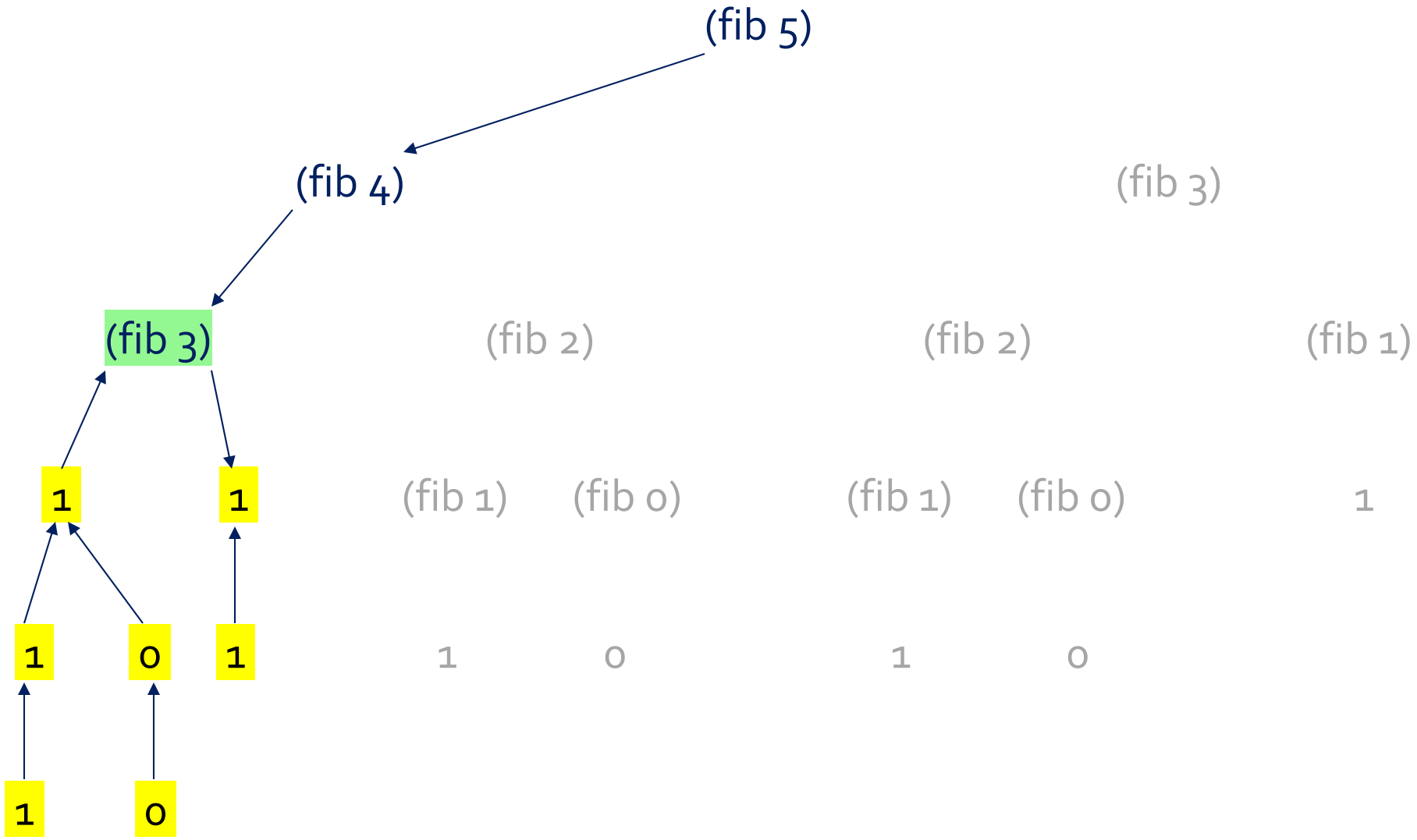

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
(fib 0)
1
1
0
1
0
1
0

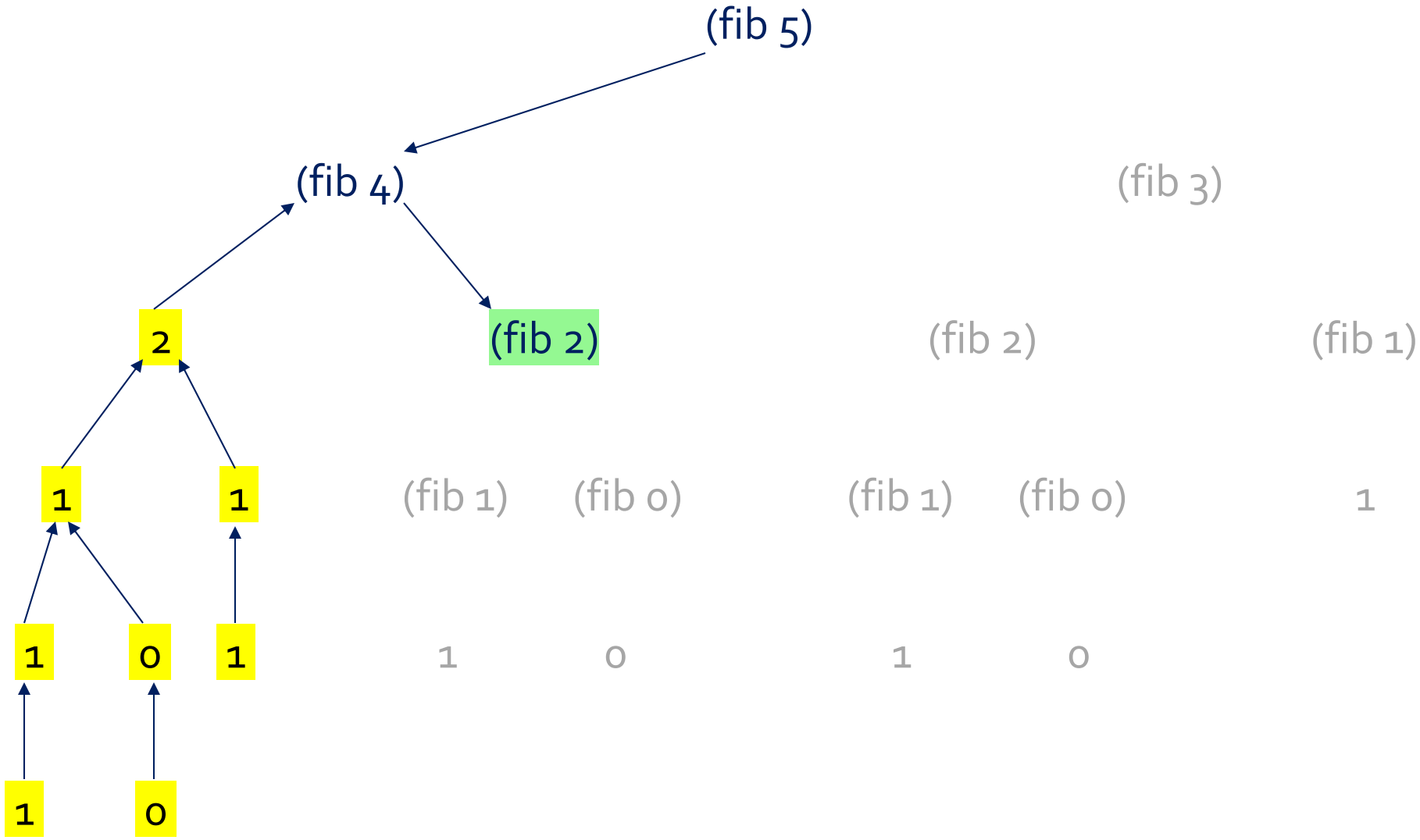

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
(fib 0)
1
1
0
1
0
1
0

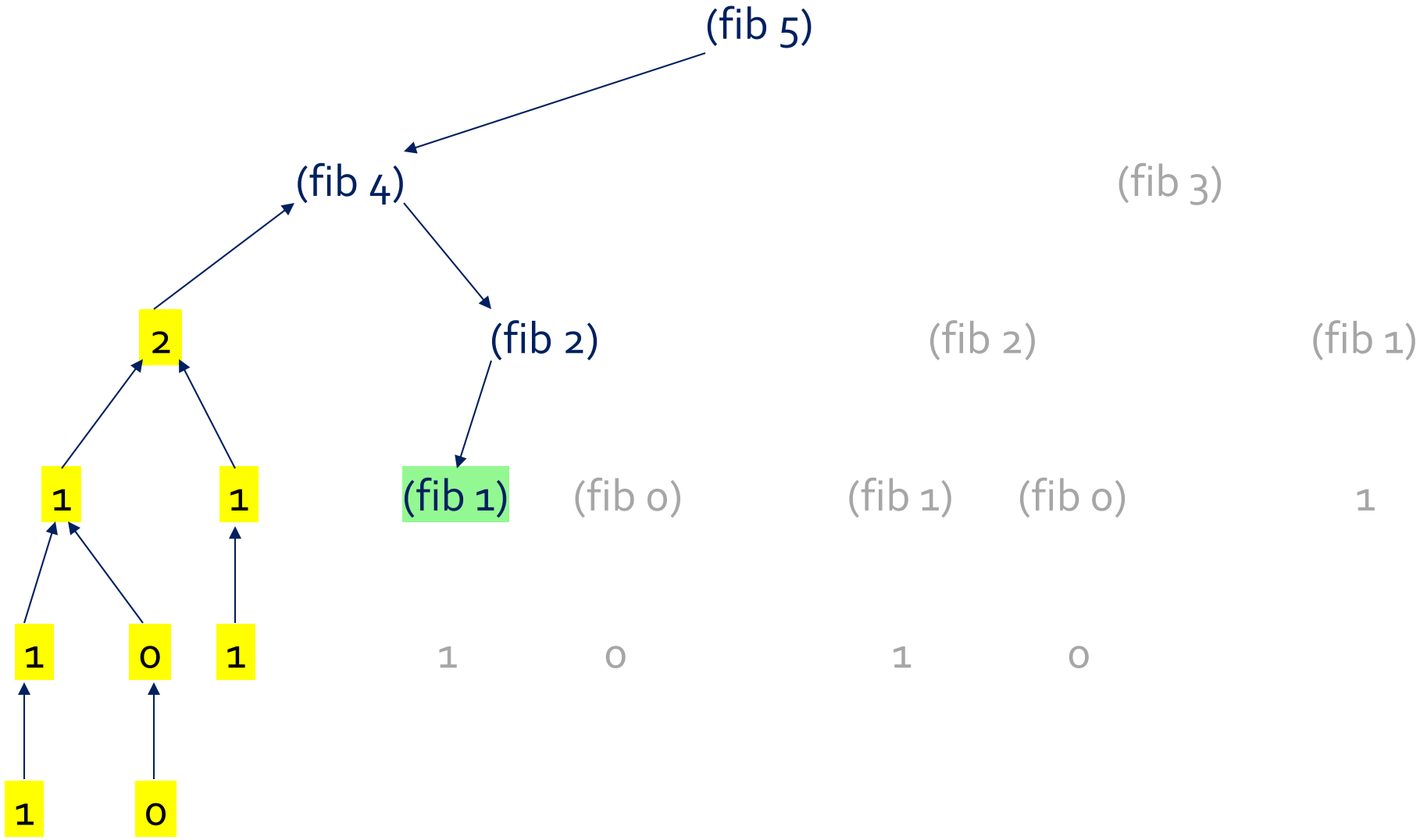

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
(fib 2)
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0

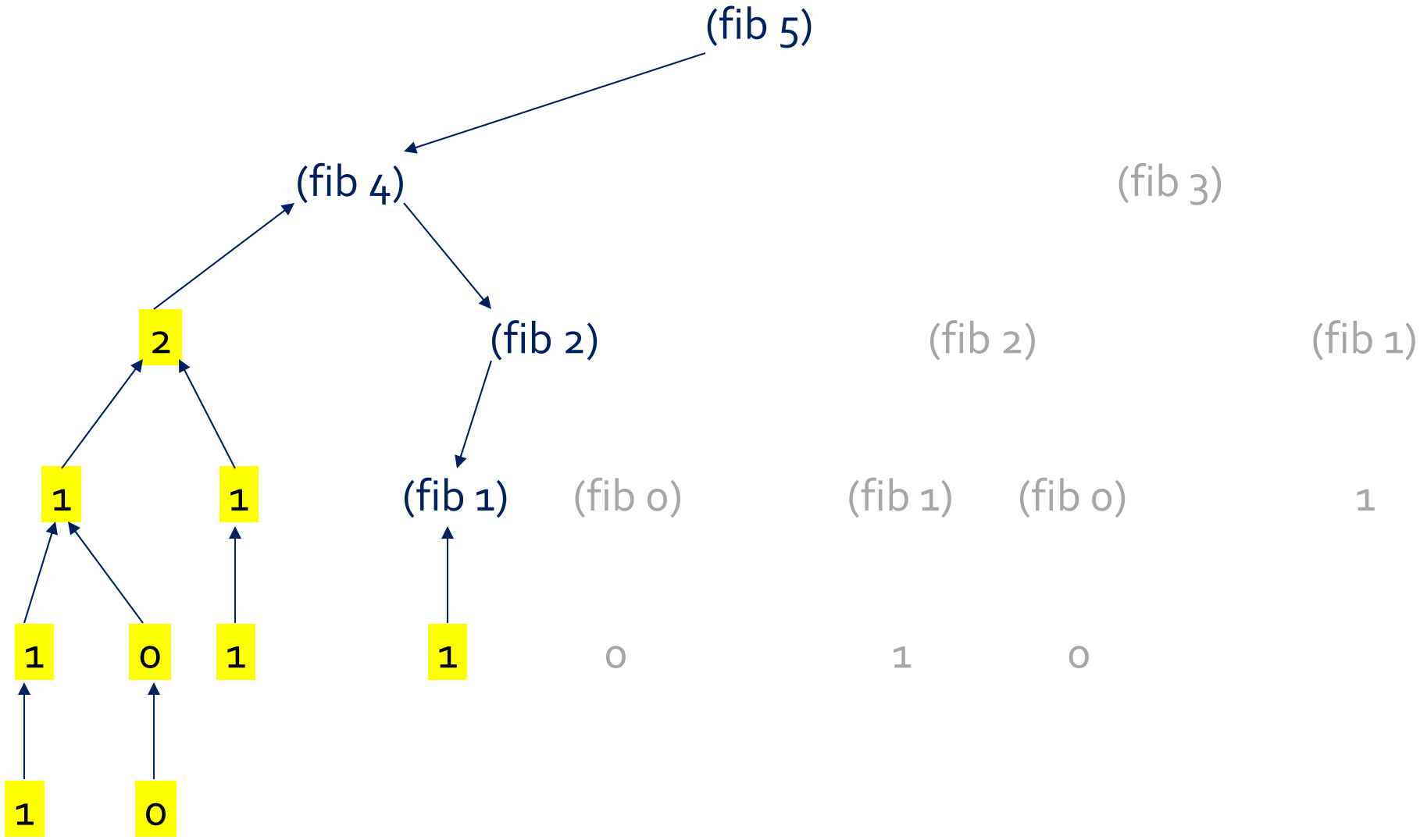

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0
1
0

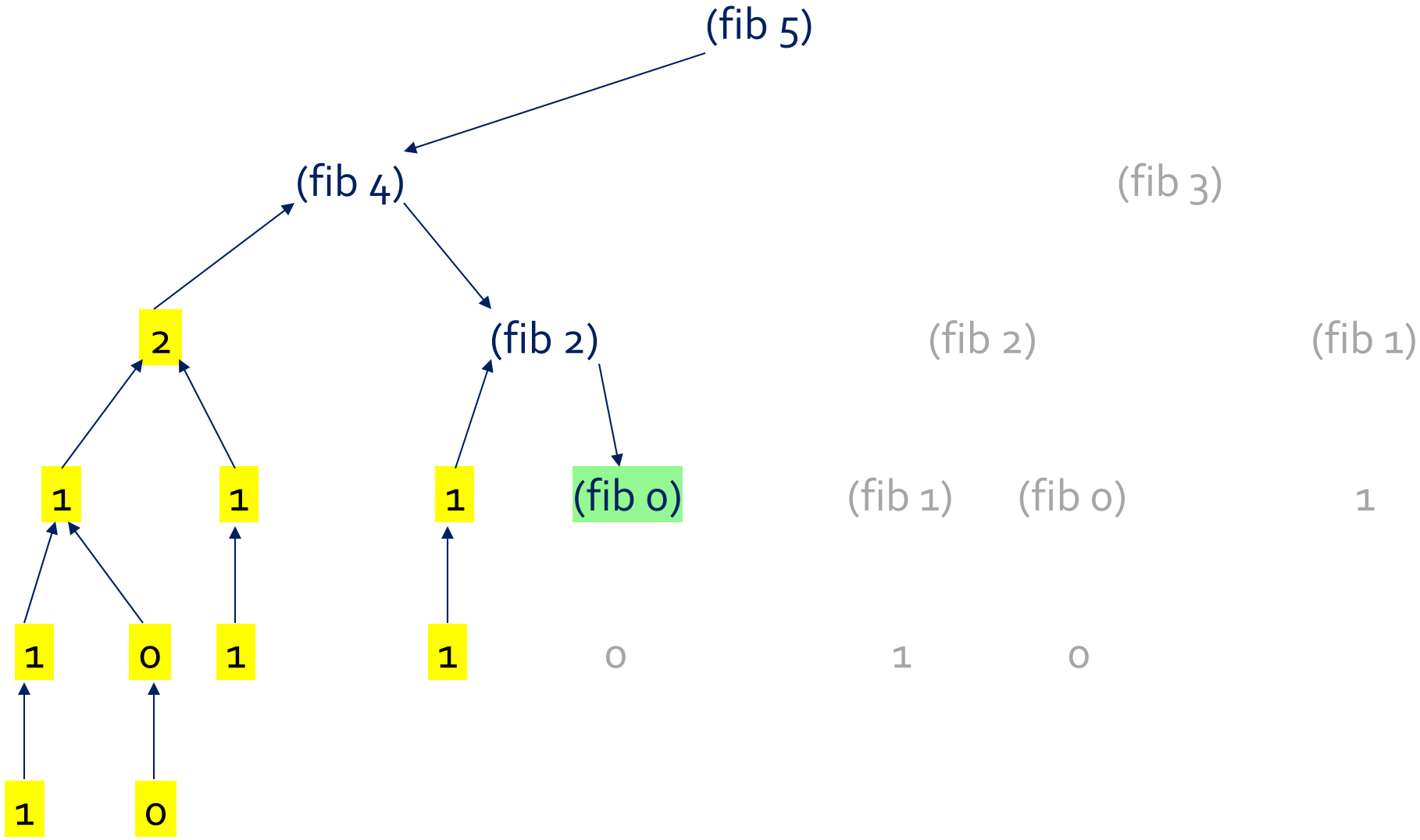

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
(fib 1)
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
0
1
0

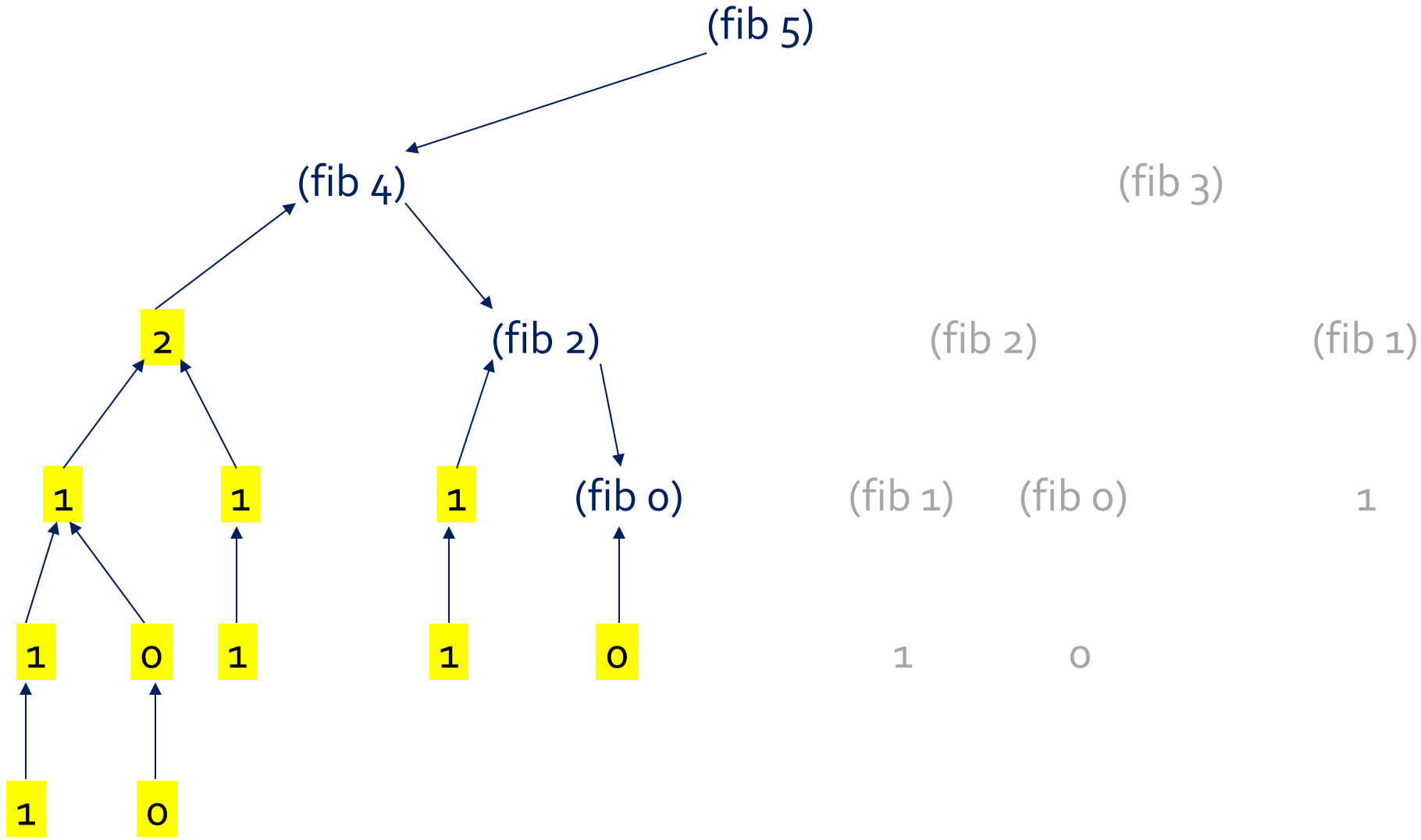

(fib 5)
(fib 4)
(fib 3)
(fib 3)
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0

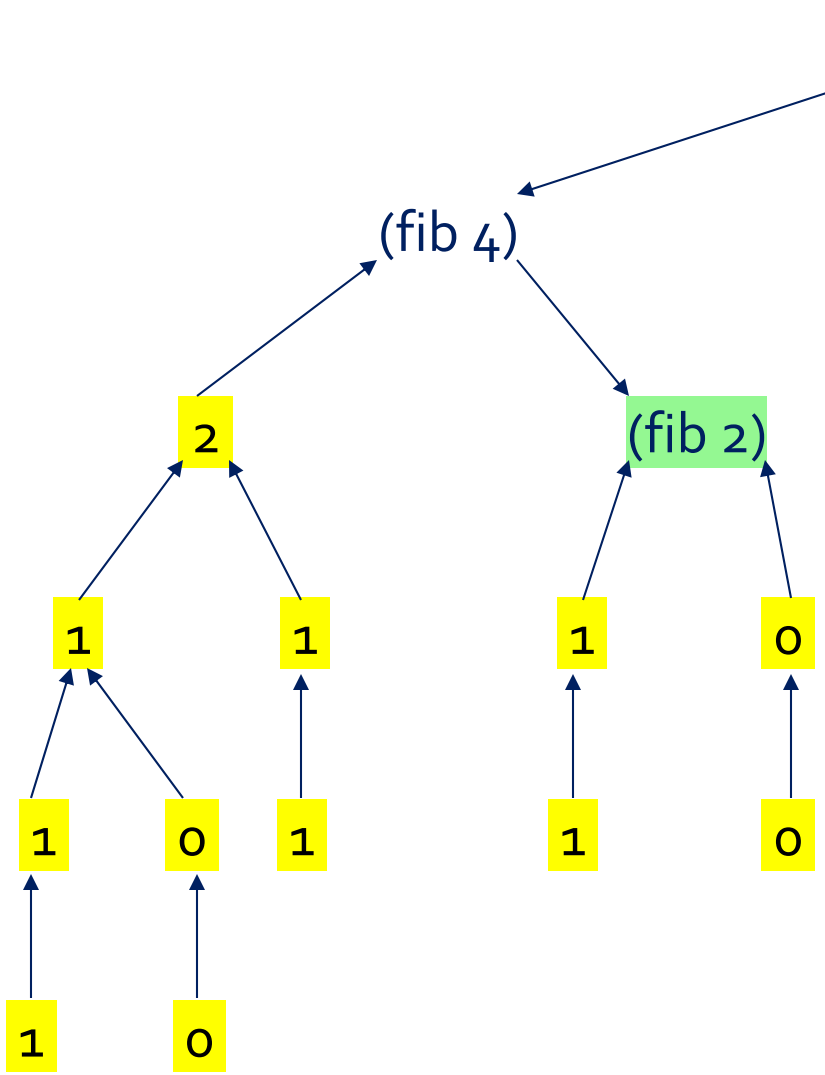

(fib 5)
(fib 4)
(fib 3)
2
(fib 2)
(fib 2)
(fib 1)
1
1
(fib 1)
(fib 0)
(fib 1)
(fib 0)
1
1
0
1
1
0
1
0


(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
1
0
1
0
1
0
1
0


(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


(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


(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




(fib 3)

(fib 2)

(fib 1)

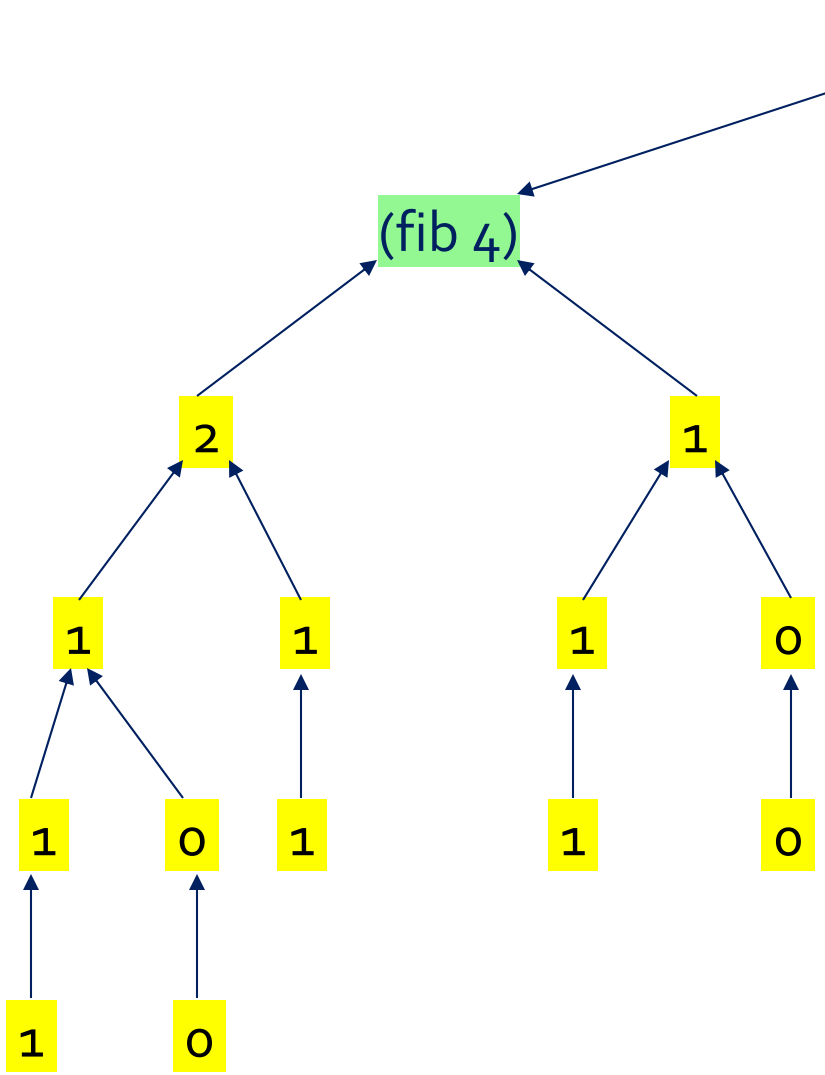
(fib 1)

(fib 0)

1

1

0

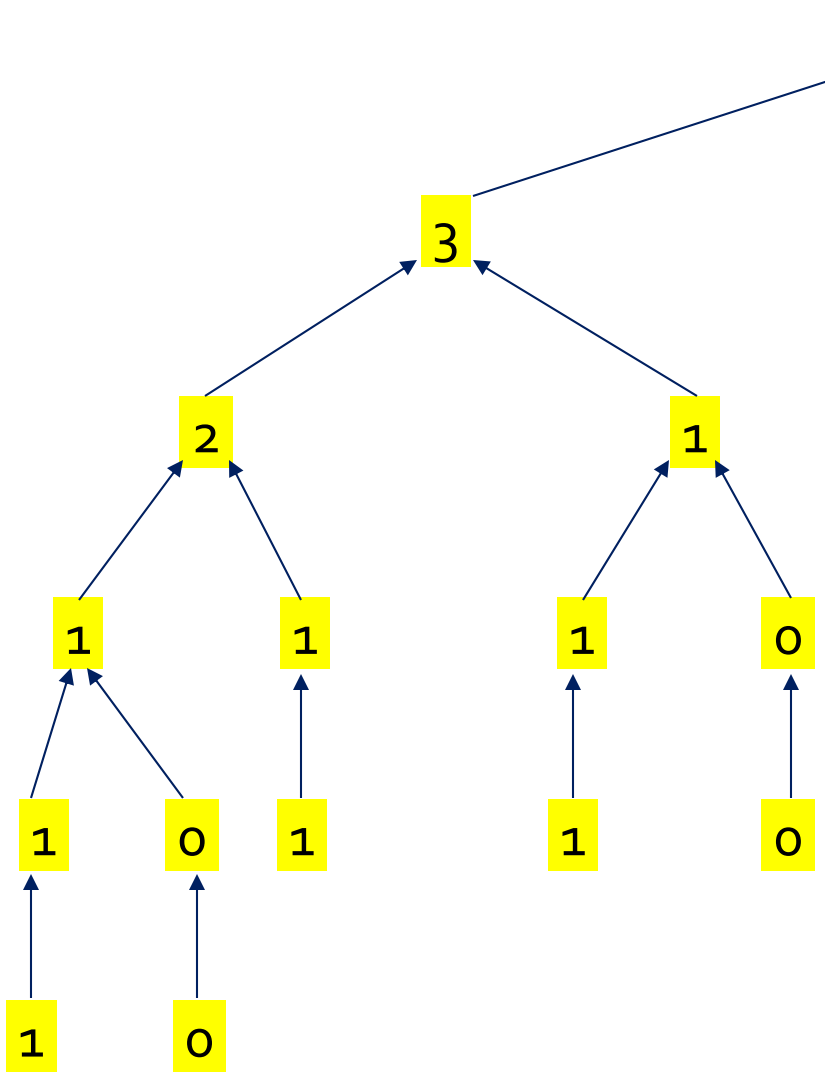



(fib 3)

(fib 2) (fib 1)

(fib 1) (fib 0) 1

1 0




(fib 3)

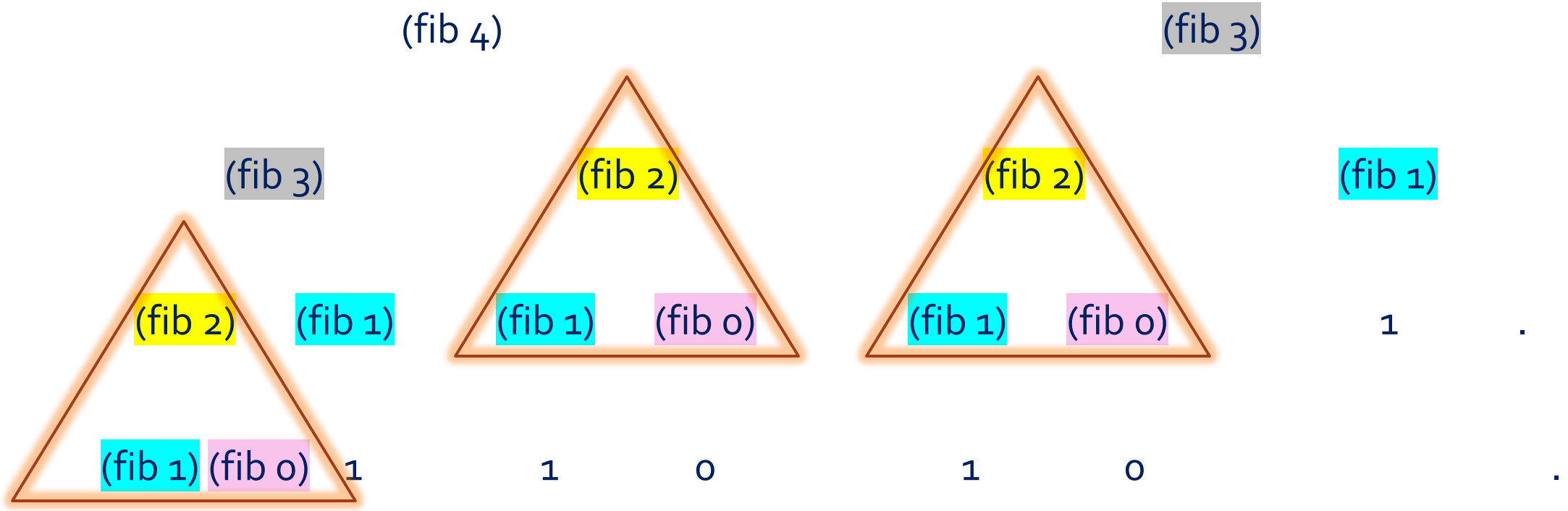
(fib 2) (fib 1)

(fib 1) (fib 0) 1

1 0

.....etc.

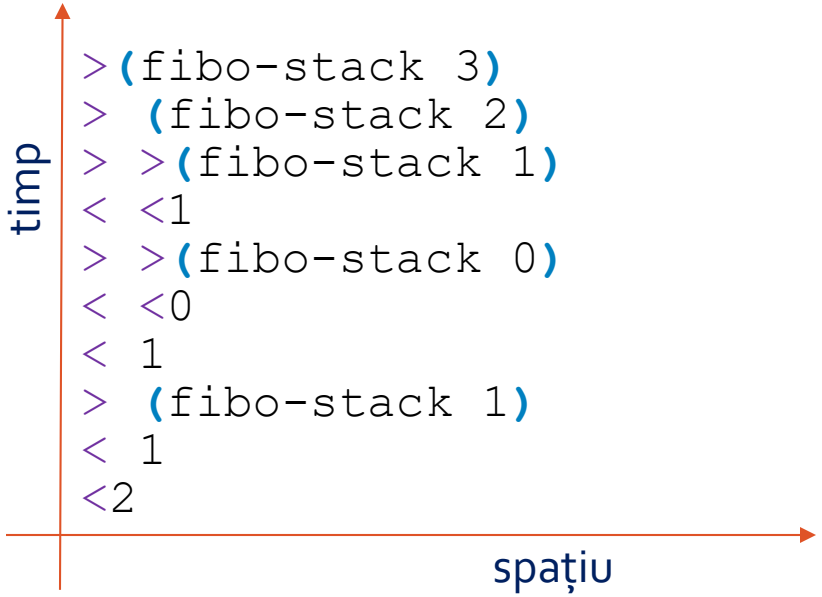
(fib 5)




← multitudine de calcule duplicate

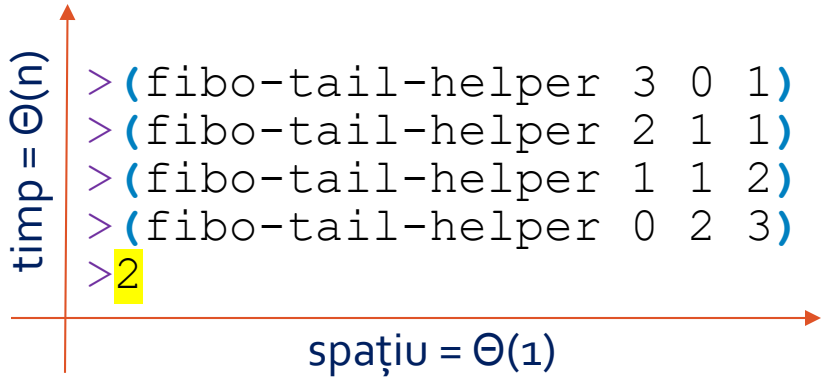
# Observații – recursivitate arborescentă

- 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



# 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))))
```

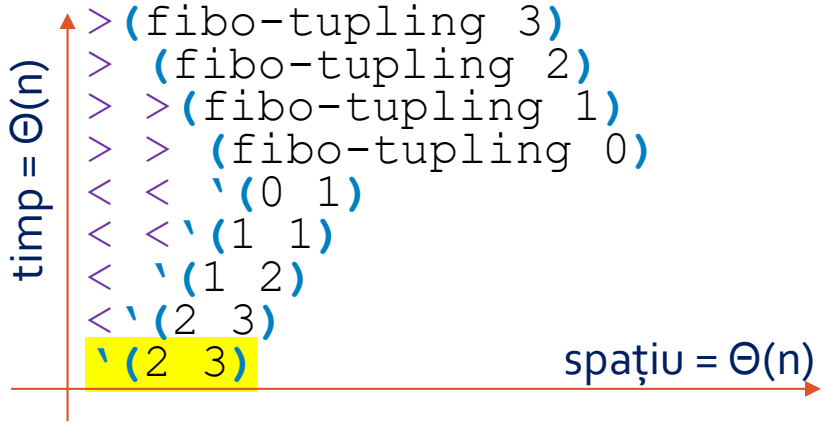



```
>(fibo-tail-helper 3 0 1)
>(fibo-tail-helper 2 1 1)
>(fibo-tail-helper 1 1 2)
>(fibo-tail-helper 0 2 3)
>2
```

← fib(0), fib(1)
← fib(1), fib(2)
← fib(2), fib(3)
← fib(3), fib(4)

# Fibonacci cu tehnica „tupling”

```
1.  (define (fibo-tupling n)
2.   (if (zero? n)
3.       (cons 0 1)
4.       (match (fibo-tupling (- n 1))
5.         [(cons fn-1 fn) (cons fn (+ fn-1 fn))])))
```



# Observații – tupling

- Tupling: tehnică de reducere a complexității temporale prin îmbogățirea datelor returnate

Utilizări

- Tupling care elimină recursivitatea arborescentă
  - Ex: fibonacci
    – un singur apel recursiv care întoarce perechea ($F_{n-1}$ . $F_n$), suficientă pentru a calcula ($F_n$ . $F_{n+1}$)
    – calculul se face la revenire (am îmbogățit rezultatul, nu argumentele)
- Tupling care elimină necesitatea parcurgerilor multiple ale aceleiași structuri
  - Ex: media aritmetică a unei liste
    – realizează o singură parcurgere care întoarce perechea (sumă . număr-de-elemente)

# 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 dacă aceleași date sunt prelucrate de mai multe noduri din arbore (adesea ineficiența este doar spațială)

Dacă există o soluție iterativă eficientă pentru problemă, aceasta se poate transpune întotdeauna 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)))))
```

# 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)))))
```

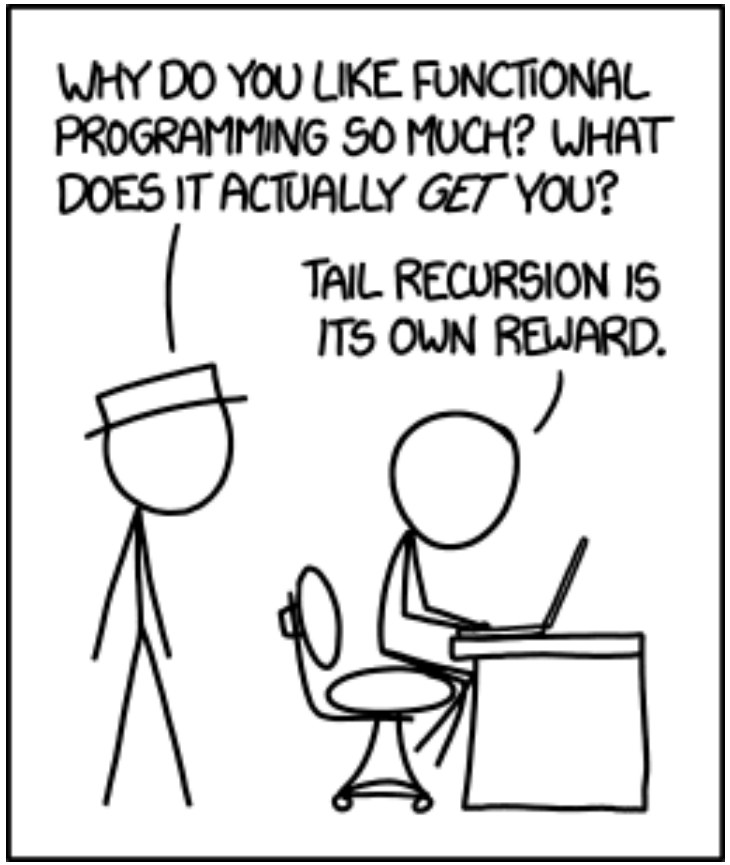
pe stivă

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))))))
```

```
>(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!

# 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: $\Theta(n^2)$ ($\Theta$(length(acc)) pentru fiecare append în parte)
  ( 0 + 1 + 2 + ... + (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))))          → 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.

# Rezumat

Teza lui Church

Recursivitate pe stivă

Recursivitate pe coadă

Recursivitate arborescentă

Tehnici de transformare

Tail-call optimization

# Rezumat

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

Recursivitate pe stivă

Recursivitate pe coadă

Recursivitate arborescentă

Tehnici de transformare

Tail-call optimization

# Rezumat

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

Recursivitate pe stivă: un apel recursiv în poziție nefinală (așteptat de alte operații), necesită salvarea contextului pe stivă – ineficientă spațial

Recursivitate pe coadă

Recursivitate arborescentă

Tehnici de transformare

Tail-call optimization

# Rezumat

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

Recursivitate pe stivă: un apel recursiv în poziție nefinală (așteptat de alte operații), necesită salvarea contextului pe stivă – ineficientă spațial

Recursivitate pe coadă: un apel recursiv în poziție finală, nu necesită salvarea contextului pe stivă – eficientă

Recursivitate arborescentă

Tehnici de transformare

Tail-call optimization

# Rezumat

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

Recursivitate pe stivă: un apel recursiv în poziție nefinală (așteptat de alte operații), necesită salvarea contextului pe stivă – ineficientă spațial

Recursivitate pe coadă: un apel recursiv în poziție finală, nu necesită salvarea contextului pe stivă – eficientă

Recursivitate arborescentă: minim două apeluri recursive lansate de apelul curent, necesită stivă și poate cauza recalcularea acelorași subprobleme – ineficientă spațial (± temporal)

Tehnici de transformare

Tail-call optimization

# Rezumat

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

Recursivitate pe stivă: un apel recursiv în poziție nefinală (așteptat de alte operații), necesită salvarea contextului pe stivă – ineficientă spațial

Recursivitate pe coadă: un apel recursiv în poziție finală, nu necesită salvarea contextului pe stivă – eficientă

Recursivitate arborescentă: minim două apeluri recursive lansate de apelul curent, necesită stivă și poate cauza recalcularea acelorași subprobleme – ineficientă spațial (± temporal)

Tehnici de transformare: acumulator (mai multe argumente), tupling (mai multe rezultate)

Tail-call optimization

# Rezumat

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

Recursivitate pe stivă: un apel recursiv în poziție nefinală (așteptat de alte operații), necesită salvarea contextului pe stivă – ineficientă spațial

Recursivitate pe coadă: un apel recursiv în poziție finală, nu necesită salvarea contextului pe stivă – eficientă

Recursivitate arborescentă: minim două apeluri recursive lansate de apelul curent, necesită stivă și poate cauza recalcularea acelorași subprobleme – ineficientă spațial (± temporal)

Tehnici de transformare: acumulator (mai multe argumente), tupling (mai multe rezultate)

Tail-call optimization: compilatorul execută apelurile pe coadă în spațiu O(1) (nu crește stiva)