Модел на средите и изчислителни процеси

Трифон Трифонов

Функционално програмиране, 2023/24 г.

11 октомври 2023 г.

Среди в Scheme

- Връзката между символите и техните оценки се записват в речник, който се нарича **среда**.
- Всеки символ има най-много една оценка в дадена среда.
- В даден момент могат да съществуват много среди.
- Символите винаги се оценяват в една конкретна среда.
- Символите могат да има различни оценки в различни среди.
- При стартиране Scheme по подразбиране работи в **глобалната среда**.
- В глобалната среда са дефинирани символи за стандартни операции и функции.

Пример за среда

- `(define a 8)`
- `r` ⟶ Грешка!
- `(define r 5)`
- `(+ r 3)` ⟶ 8
- `(define (f x) (* x r))`
- `(f 3)` ⟶ 15
- `(f r)` ⟶ 25

E		
a : 8		
r : 5		
f :	Параметри : x	
	Тяло : (* x r)	
	Среда : E	

Функции и среди

- Всяка функция `f` пази указател към средата E, в която е дефинирана.
- При извикване на `f`:
  - създава се нова среда E_1, която разширява E
  - в E_1 всеки символ означаващ формален параметър се свързва с оценката на фактическия параметър
  - тялото на f се оценява в E_1

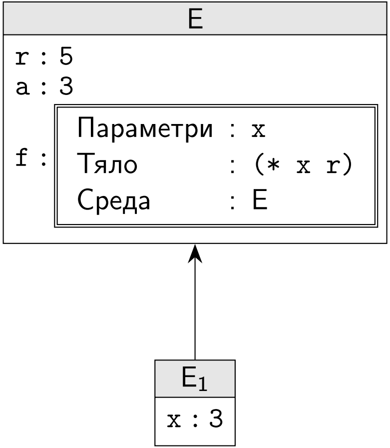
Дърво от среди

- Всяка среда пази указател към своя “родителска среда”, която разширява
- така се получава дърво от среди
- при оценка на символ в дадена среда E
    - първо се търси оценката му в E
    - ако символът не е дефиниран в E, се преминава към родителската среда
    - при достигане на най-горната среда, ако символът не е дефиниран и в нея се извежда съобщение за грешка

Извикване на дефинирана функция

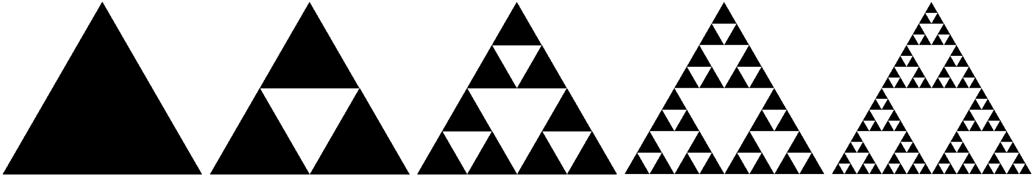
- `(define r 5)`
- `(define a 3)`
- `(define (f x) (* x r))`

```
{E}      (f a)
           ↓
{E}      (f 3)
           ↓
{E₁}    (* x r)
           ↓
          15
```



Какво е рекурсия?

Какво е рекурсия?



"Sierpinski triangle, the evolution in five iterations" от Solkoll, Обществено достояние чрез Общомедия

Какво е рекурсия?

Повторение чрез позоваване на себе си

Рекурсивна функция: дефинира се чрез себе си

$$n! = \begin{cases} 1, & \text{при } n = 0, \qquad \textbf{(база)} \\ n \cdot (n-1)!, & \text{при } n > 0. \qquad \textbf{(стъпка)} \end{cases}$$

- Дава се отговор на най-простата задача (база, дъно)
- Показва се как сложна задача се свежда към една или няколко по-прости задачи от същия вид (стъпка)

Рекурсивни уравнения

Какво означава “да дефинираме функция чрез себе си”?

Да разгледаме *рекурсивното уравнение*, в което F е неизвестно:

$$F(n) = \underbrace{\begin{cases} 1, & \text{при } n = 0, \\ n \cdot F(n-1), & \text{при } n > 0. \end{cases}}_{\Gamma(F)(n)}$$

$n!$ е “най-малкото” решение на уравнението $F = \Gamma(F)$.

Най-малка неподвижна точка

Теорема (Knaster-Tarski)

*Ако Γ е изчислим оператор, то уравнението $F = \Gamma(F)$ има единствено най-малко решение f (**най-малка неподвижна точка на** Γ). Нещо повече, решението точно съответства на рекурсивна програма пресмятаща f чрез Γ.*

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))
```

Кое е **най-малкото решение** на уравнението $F(x) = F(x+1) - 1$?

```
(define (g x) (- 1 (g (+ x 1))))
```

`(f 0)` $\longrightarrow$?

g е "празната функция", т.е. $\mathrm{dom}(g) = \emptyset$.

Операционна и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

Нека `(define (f x) Γ[f])` е рекурсивно дефинирана функция.

Коя е математическата функция f, която се пресмята от `f`?

Денотационна семантика

f е най-малката неподвижна точка на уравнението $F = \Gamma(F)$.

Операционна семантика

Разглеждаме редицата от последователни оценки на комбинации

`(f a)` $\rightarrow$ `Γ[f][x` $\mapsto$ `a]` $\rightarrow \ldots$

Ако стигнем до елемент `b`, който не е комбинация, то $f(a) := b$.

Функциите в Scheme имат дуален, но еквивалентен смисъл:

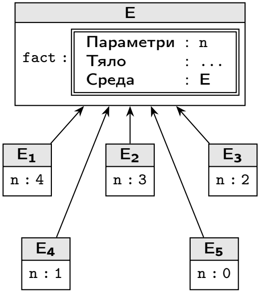
- решения на рекурсивни уравнения
- изчислителни процеси, генериращи се при оценка

Оценка на рекурсивна функция

```
(fact 4)
    ↓
(* 4 (fact 3))
    ↓
(* 4 (* 3 (fact 2)))
    ↓
(* 4 (* 3 (* 2 (fact 1))))
    ↓
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
    ↓
(* 4 (* 3 (* 2 (* 1 1))))
    ↓
(* 4 (* 3 (* 2 1)))
    ↓
(* 4 (* 3 2))
    ↓
(* 4 6)
    ↓
24
```

Оценка на рекурсивна функция в среда

{E}	`(fact 4)`
{E_1}	`(* 4 (fact 3))`
{E_2}	`(* 4 (* 3 (fact 2)))`
{E_3}	`(* 4 (* 3 (* 2 (fact 1))))`
{E_4}	`(* 4 (* 3 (* 2 (* 1 (fact 0)))))`
{E_5}	`(* 4 (* 3 (* 2 (* 1 1))))`
{E_3}	`(* 4 (* 3 (* 2 1)))`
{E_2}	`(* 4 (* 3 2))`
{E_1}	`(* 4 6)`
{E}	24



Линеен рекурсивен процес

Факториел с цикъл

Факториел на C++

```
int fact(int n) {
  int r = 1;
  for( int i = 1 ; i <= n ; i++ )
    r *= i;
  return r;
}
```

Превод на Scheme

```
(define (for n r i )
  (if (<= i n)
      (for n (* r i) (+ i 1) )
      r ))

(define (fact n)
  (for n 1 1 ))
```

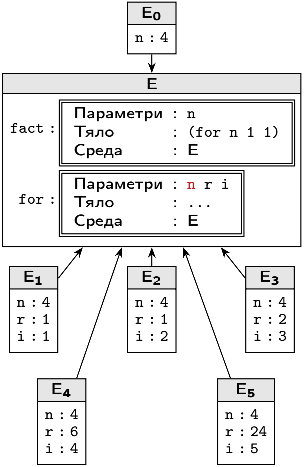
Оценка на итеративен факториел

$$
\begin{array}{c}
\texttt{(fact 4)} \\
\downarrow \\
\texttt{(for 4 1 1)} \\
\downarrow \\
\texttt{(for 4 1 2)} \\
\downarrow \\
\texttt{(for 4 2 3)} \\
\downarrow \\
\texttt{(for 4 6 4)} \\
\downarrow \\
\texttt{(for 4 24 5)} \\
\downarrow \\
24
\end{array}
$$

Линеен итеративен процес

Оценка на итеративен факториел със среди

{E}	`(fact 4)`
{E_0}	`(for 4 1 1)`
{E_1}	`(for 4 1 2)`
{E_2}	`(for 4 2 3)`
{E_3}	`(for 4 6 4)`
{E_4}	`(for 4 24 5)`
{E_5}	24



Рекурсивен и итеративен процес

```
(fact 4)
↓
(* 4 (fact 3))
↓
(* 4 (* 3 (fact 2)))
↓
(* 4 (* 3 (* 2 (fact 1))))
↓
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
↓
(* 4 (* 3 (* 2 (* 1 1))))
↓
(* 4 (* 3 (* 2 1)))
↓
(* 4 (* 3 2))
↓
(* 4 6)
↓
24
```

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))
```

```
(fact 4)
↓
(for 4 1 1)
↓
(for 4 1 2)
↓
(for 4 2 3)
↓
(for 4 6 4)
↓
(for 4 24 5)
↓
24
```

```
(define (for n r i)
  (if (<= i n)
      (for n (* r i) (+ i 1))
      r))

(define (fact n)
  (for n 1 1))
```

Опашкова рекурсия

- Функциите, в които има отложени операции генерират същински **рекурсивни процеси**
- Рекурсивно извикване, при което няма отложена операция се нарича **опашкова рекурсия**
- Функциите, в които всички рекурсивни извиквания са опашкови генерират **итеративни процеси**
- При итеративните процеси липсва етап на свиването на рекурсията
- Опашковата рекурсия се използва за симулиране на цикли
- В Scheme опашковата рекурсия по стандарт се интерпретира като цикъл
  - т.е. не се заделя памет за всяко рекурсивно извикване

Рекурсивен и итеративен процес

```
(fact 4)
↓
(* 4 (fact 3))
↓
(* 4 (* 3 (fact 2)))
↓
(* 4 (* 3 (* 2 (fact 1))))
↓
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
↓
(* 4 (* 3 (* 2 (* 1 1))))
↓
(* 4 (* 3 (* 2 1)))
↓
(* 4 (* 3 2))
↓
(* 4 6)
↓
24
```

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1)))))
```

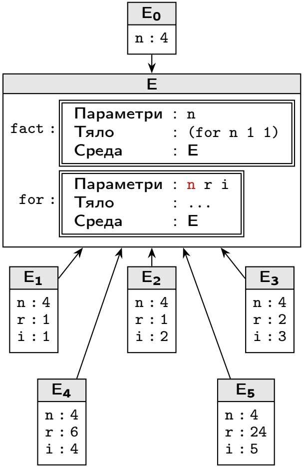
```
(fact 4)
↓
(for 4 1 1)
↓
(for 4 1 2)
↓
(for 4 2 3)
↓
(for 4 6 4)
↓
(for 4 24 5)
↓
24
```

```
(define (for n r i)
  (if (<= i n)
      (for n (* r i) (+ i 1))
      r))

(define (fact n)
  (for n 1 1))
```

Оценка на итеративен факториел със среди

{E}	`(fact 4)`
	↓
{E_0}	`(for 4 1 1)`
	↓
{E_1}	`(for 4 1 2)`
	↓
{E_2}	`(for 4 2 3)`
	↓
{E_3}	`(for 4 6 4)`
	↓
{E_4}	`(for 4 24 5)`
	↓
{E_5}	24



Вложени дефиниции

- (define (<функция> {<параметър>}) {<дефиниция>} <тяло>)
- При извикване на <функция> първо се оценяват всички <дефиниция> и след това се оценява <тяло>
  - Първо се създава среда E_1, в която формалните параметри се свързват с оценките на фактическите
  - След това се създава среда E_2, която разширява E_1, за вложените дефиниции
  - В средата E_2 се записват всички символи от вложени дефиниции без стойности
  - Всички вложени дефиниции се **оценяват** в E_2
  - Накрая получените оценки се **свързват** със съответните си символи в E_2
- **Пример:**

```
(define (dist x1 y1 x2 y2)
  (define dx (- x2 x1))
  (define dy (- y2 y1))
  (define (sq x) (* x x))
  (sqrt (+ (sq dx) (sq dy))))
```

Оценка на вложени функции

$\{E\}$ `(dist 2 5 -1 9)`

↓

$\{E_2\}$ `(define dx (- x2 x1))`

$\{E_2\}$ `(define dy (- y2 y1))`

$\{E_2\}$ `(define (sq x) (* x x))`

$\{E_2\}$ `(sqrt (+ (sq dx) (sq dy)))`

↓

$\{E_3\}$ `(sqrt (+ (* x x) (sq dy)))`

↓

$\{E_4\}$ `(sqrt (+ 9 (* x x)))`

↓

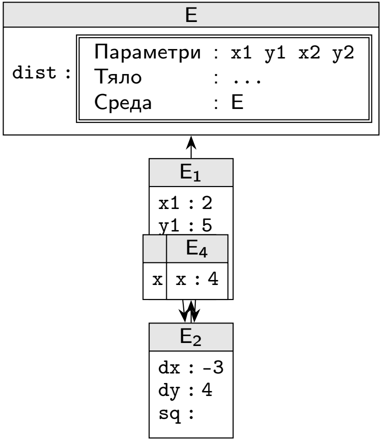
$\{E_2\}$ `(sqrt (+ 9 16))`

↓

$\{E_2\}$ `(sqrt 25)`

↓

$\{E_2\}$ `5`



Вложена помощна итеративна функция

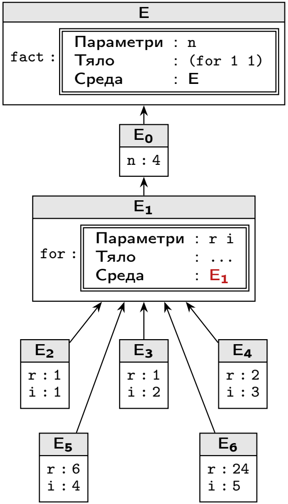
При итеративни функция е удобно помощната функция да е вложена.

```
(define (fact n)
  (define (for r i)
    (if (<= i n)
        (for (* r i) (+ i 1))
        r))
  (for 1 1))
```

Вложените дефиниции "виждат" символите на обхващащите им дефиниции.

Оценка на итеративен факториел с вложена функция

{E} `(fact 4)`
↓
{E_1} `(define (for r i)...)`
↓
{E_1} `(for 1 1)`
↓
{E_2} `(for 1 2)`
↓
{E_3} `(for 2 3)`
↓
{E_4} `(for 6 4)`
↓
{E_5} `(for 24 5)`
↓
{E_6} 24



Специална форма let

- ```
  (let ({(<символ> <израз>)}) <тяло>)
  ```
- ```
  (let ((<символ₁> <израз₁>)
        (<символ₂> <израз₂>)
        ...
        (<символₙ> <изразₙ>))
    <тяло>)
  ```
- При оценка на `let` в среда E:
  - Създава се нова среда E_1 разширение на текущата среда E
  - Оценката на <израз$_1$> в E се свързва със <символ$_1$> в E_1
  - Оценката на <израз$_2$> в E се свързва със <символ$_2$> в E_1
  - ...
  - Оценката на <израз$_n$> в E се свързва със <символ$_n$> в E_1
  - Връща се оценката на <тяло> в средата E_1
- `let` няма странични ефекти върху средата!

Пример за let

```
(define (dist x1 y1 x2 y2)
  (let ((dx (- x2 x1))
        (dy (- y2 y1)))
   (sqrt (+ (sq dx) (sq dy)))))

(define (area x1 y1 x2 y2 x3 y3)
  (let ((a (dist x1 y1 x2 y2))
        (b (dist x2 y2 x3 y3))
        (c (dist x3 y3 x1 y1))
        (p (/ (+ a b c) 2)))
   (sqrt (* p (- p a) (- p b) (- p c)))))
```

Оценка на let

{E} `(dist 2 5 -1 9)`

↓

`(let ((dx (- x2 x1))`

{E_1} `(dy (- y2 y1)))`

`(sqrt (+ (sq dx) (sq dy))))`

↓

{E_2} `(sqrt (+ (sq dx) (sq dy)))`

↓

{E_3} `(sqrt (+ (* x x) (sq dy)))`

↓

{E_4} `(sqrt (+ 9 (* x x)))`

↓

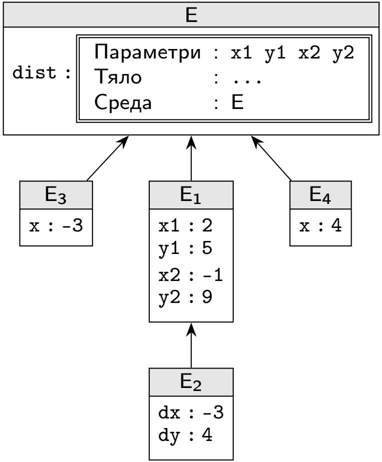
{E_2} `(sqrt (+ 9 16))`

↓

{E_2} `(sqrt 25)`

↓

{E_2} `5`



Специална форма let*

- (let* ({(<символ> <израз>)}) <тяло>)
- ```
  (let* ((<символ₁> <израз₁>)
         (<символ₂> <израз₂>)
         ...
         (<символₙ> <изразₙ>))
    <тяло>)
  ```
- При оценка на let* в среда E:
  - Създава се нова среда E_1 разширение на текущата среда E
  - Оценката на <$израз_1$> в E се свързва със <$символ_1$> в E_1
  - Създава се нова среда E_2 разширение на текущата среда E_1
  - Оценката на <$израз_2$> в E_1 се свързва със <$символ_2$> в E_2
  - ...
  - Създава се нова среда E_n разширение на текущата среда E_{n-1}
  - Оценката на <$израз_n$> в E_{n-1} се свързва със <$символ_n$> в E_n
  - Връща се оценката на <тяло> в средата E_n
  ```

Пример за let*

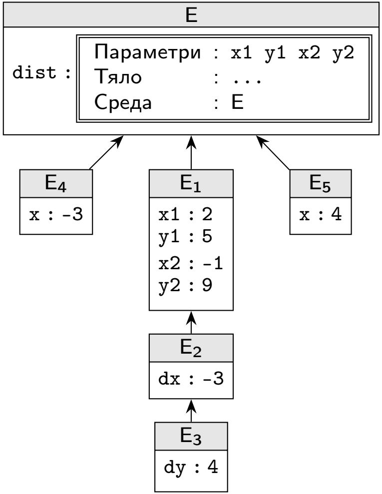
```
(define (area x1 y1 x2 y2 x3 y3)
  (let* ((a (dist x1 y1 x2 y2))
         (b (dist x2 y2 x3 y3))
         (c (dist x3 y3 x1 y1))
         (p (/ (+ a b c) 2)))
```

Редът има значение!

```
(define (area x1 y1 x2 y2 x3 y3)
  (let* ((p (/ (+ a b c) 2))
         (a (dist x1 y1 x2 y2))
         (b (dist x2 y2 x3 y3))
         (c (dist x3 y3 x1 y1)))
    (sqrt (* p (- p a) (- p b) (- p c)))))
```

Оценка на let*

```
{E}       (dist 2 5 -1 9)
             ↓
          (let* ((dx (- x2 x1))
{E1}             (dy (- y2 y1)))
            (sqrt (+ (sq dx) (sq dy))))
             ↓
{E3}      (sqrt (+ (sq dx) (sq dy)))
             ↓
{E4}      (sqrt (+ (* x x) (sq dy)))
             ↓
{E5}      (sqrt (+ 9 (* x x)))
             ↓
{E3}      (sqrt (+ 9 16))
             ↓
{E3}      (sqrt 25)
             ↓
{E3}      5
```



Степенуване

Функцията x^n може да се дефинира по следния начин:

$$x^n = \begin{cases} 1, & \text{ако } n = 0, \\ \frac{1}{x^{-n}}, & \text{ако } n < 0, \\ x \cdot x^{n-1}, & \text{ако } n > 0. \end{cases}$$

```
(define (pow x n)
  (cond ((= n 0) 1)
        ((< n 0) (/ 1 (pow x (- n))))
        (else (* x (pow x (- n 1))))))
```

Оценка на степенуване

```
(pow 2 6)
↓
(* 2 (pow 2 5))
↓
(* 2 (* 2 (pow 2 4)))
↓
(* 2 (* 2 (* 2 (pow 2 3))))
↓
(* 2 (* 2 (* 2 (* 2 (pow 2 2)))))
↓
(* 2 (* 2 (* 2 (* 2 (* 2 (pow 2 1))))))
↓
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (pow 2 0)))))))
↓
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 1))))))
↓
(* 2 (* 2 (* 2 (* 2 (* 2 2)))))
↓
(* 2 (* 2 (* 2 (* 2 4))))
↓
(* 2 (* 2 (* 2 8)))
↓
(* 2 (* 2 16))
↓
(* 2 32)
↓
64
```

Линеен рекурсивен процес

Бързо степенуване

Алтернативна дефиниция на x^n:

$$x^n = \begin{cases} 1, & \text{ако } n = 0, \\ \frac{1}{x^{-n}}, & \text{ако } n < 0, \\ (x^{\frac{n}{2}})^2, & \text{ако } n > 0, n \text{ — четно}, \\ x \cdot x^{n-1}, & \text{ако } n > 0, n \text{ — нечетно}. \end{cases}$$

```scheme
(define (qpow x n)
  (define (sqr x) (* x x))
  (cond ((= n 0) 1)
        ((< n 0) (/ 1 (qpow x (- n))))
        ((even? n) (sqr (qpow x (quotient n 2))))
        (else (* x (qpow x (- n 1))))))
```

Оценка на бързо степенуване

```
(qpow 2 6)
    ↓
(sqr (qpow 2 3))
    ↓
(sqr (* 2 (qpow 2 2)))
    ↓
(sqr (* 2 (sqr (qpow 2 1))))
    ↓
(sqr (* 2 (sqr (* 2 (qpow 2 0)))))
    ↓
(sqr (* 2 (sqr (* 2 1))))
    ↓
(sqr (* 2 (sqr 2)))
    ↓
(sqr (* 2 4))
    ↓
(sqr 8)
    ↓
64
```

Логаритмичен рекурсивен процес

Числа на Фибоначи

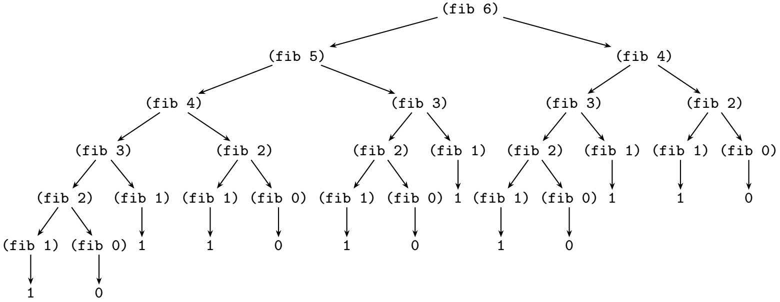
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$

$$f_n = \begin{cases} 0, & \text{за } n = 0, \\ 1, & \text{за } n = 1, \\ f_{n-1} + f_{n-2}, & \text{за } n \geq 2. \end{cases}$$

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))
```

$f_{40} = ?$

Дървовидна рекурсия



Дървовиден рекурсивен процес

Как да оптимизираме?

Решение №1: мемоизация

Да помним вече пресметнатите стойности, вместо да ги смятаме пак.

За ефективна реализация обикновено са нужни странични ефекти.

Решение №2: динамично програмиране

Строим последователно всички числа на Фибоначи в нарастващ ред.

Нужно е да помним само последните две числа!

```
(define (fib n)
  (define (iter i fi fi-1)
    (if (= i n) fi
        (iter (+ i 1) (+ fi fi-1) fi)))
  (if (= n 0) 0
      (iter 1 1 0)))
```

Итеративно генериране на числата на Фибоначи

```
   (fib 7)
      ↓
(iter 1 1 0)
      ↓
(iter 2 1 1)
      ↓
(iter 3 2 1)
      ↓
(iter 4 3 2)
      ↓
(iter 5 5 3)
      ↓
(iter 6 8 5)
      ↓
(iter 7 13 8)
      ↓
     13
```