

# Модел на средите и изчислителни процеси

Трифон Трифонов

Функционално програмиране, 2017/18 г.

19 октомври 2017 г.

## Среди в Scheme

- Връзката между символите и техните оценки се записват в речник, който се нарича **среда**.

## 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.

## 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.

## 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.

## 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.

## 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**.

## 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`)

E
a : 8

# Пример за среда

- (`define a 8`)
- `r` → Грешка!

E
a : 8

# Пример за среда

- (`define a 8`)
- `r` → Грешка!
- (`define r 5`)

E
a : 8
r : 5

# Пример за среда

- (`define a 8`)
- `r` → Грешка!
- (`define r 5`)
- (`+ r 3`) → 8

E
a : 8
r : 5

# Пример за среда

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



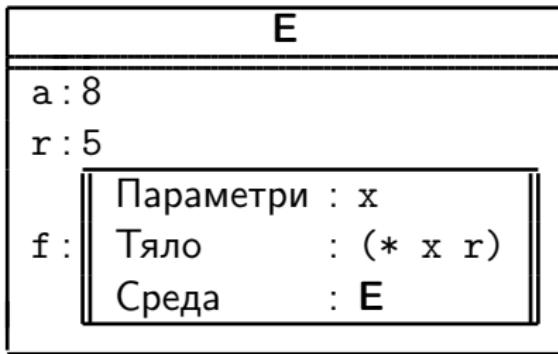
# Пример за среда

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



# 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



# Функции и среди

- Всяка функция  $f$  пази указател към средата **E**, в която е дефинирана.

# Функции и среди

- Всяка функция  $f$  пази указател към средата **E**, в която е дефинирана.
- При извикване на  $f$ :

# 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  $E$

# 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

# 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

## Дърво от среди

- Всяка среда пази указател към своя “родителска среда”, която разширява
- така се получава дърво от среди

## 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**

## 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**

## 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

## 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)

E
r : 5

# Извикване на дефинирана функция

- (`define r 5`)
- (`define a 3`)

E
r : 5
a : 3

# Извикване на дефинирана функция

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

E	
r : 5	
a : 3	
f :	Параметри : x Тяло : (* x r) Среда : E

# Извикване на дефинирана функция

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

<b>E</b>	
r : 5	
a : 3	
f :	Параметри : x Тяло : (* x r) Среда : <b>E</b>

# Извикване на дефинирана функция

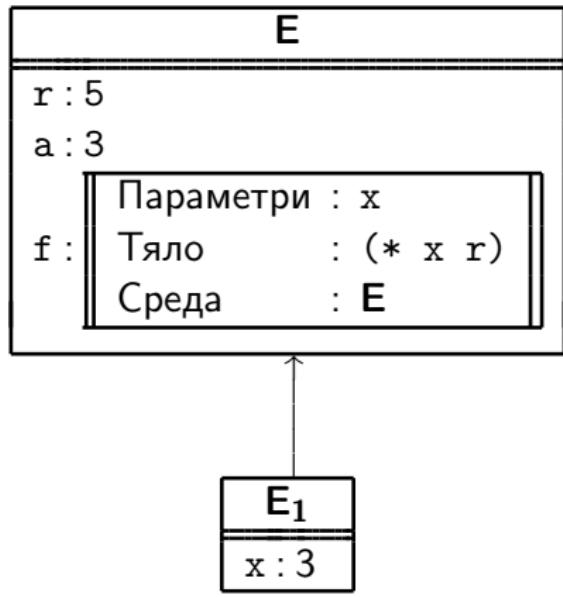
- (`define r 5`)
- (`define a 3`)
- (`define (f x) (* x r)`)
- {**E**}      (f a)
 

↓
- {**E**}      (f 3)

<b>E</b>	
r : 5	
a : 3	
f :	Параметри : x Тяло : (* x r) Среда : <b>E</b>

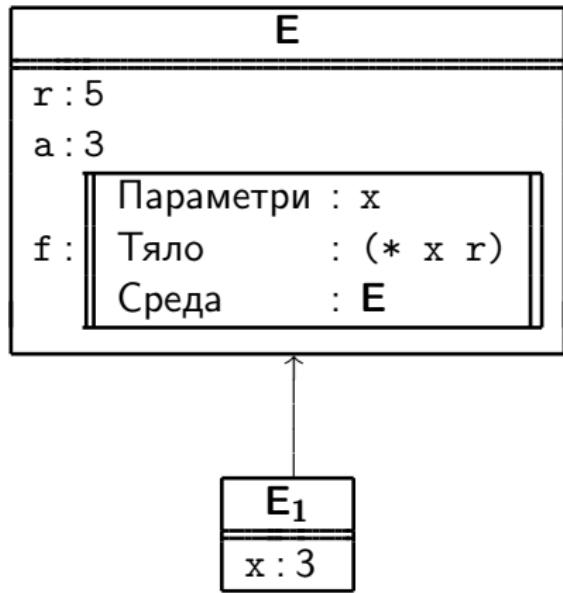
# Извикване на дефинирана функция

- (`define r 5`)
- (`define a 3`)
- (`define (f x) (* x r)`)
- {`E`}      (`f a`)
  - ↓
  - {`E`}      (`f 3`)
    - ↓
    - {`E1`}    (`(* x r)`)



# Извикване на дефинирана функция

- (`define r 5`)
- (`define a 3`)
- (`define (f x) (* x r)`)
- {E}      (f a)
  - ↓
  - {E}      (f 3)
    - ↓
    - {E<sub>1</sub>}    (\* x r)
      - ↓
      - 15



# Какво е рекурсия?

GNU = GNU is Not Unix

# Какво е рекурсия?



# Какво е рекурсия?



# Какво е рекурсия?

# Какво е рекурсия?

Повторение чрез позоваване на себе си

# Какво е рекурсия?

Повторение чрез позоваване на себе си

Рекурсивна функция: дефинира се чрез себе си

$$! : \mathbb{N} \rightarrow \mathbb{N}$$

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

31

$$f(n) = l$$

$$f: A \rightarrow B$$

$$f(a) = b_1 \wedge f(u) = b_2 \rightarrow b_1 = b_2.$$

$$(a, b) \in F$$

$$F \subseteq A \times B$$

$$\boxed{\begin{array}{l} (a, b_1) \in F \wedge (a, b_2) \in F \\ \Rightarrow b_1 = b_2 \end{array}} \quad \leq 1$$

Ологнүүчүүлүмдөр :  $\forall a \in A \forall b_{1,2} \in B$

$$(a, b_1) \in F \wedge (a, b_2) \in F \rightarrow b_1 = b_2$$

ТОТМНОСТ :  $\forall a \in A \exists b \in B (a, b) \in F$

Узүүгүй :  $\forall a_{1,2} \in A \forall b \in B$

$$(a_1, b) \in F \wedge (a_2, b) \in F \rightarrow a_1 = a_2$$

Чоңуук :  $\forall b \in B \exists a \in A (a, b) \in F$

$F$  е чөнүүлүк  $\Leftrightarrow F^{-1}$  е ологнүүлүктөш

$F$  е чоңуук  $\Leftrightarrow F^{-1}$  е ТОТМНОСТ

# Какво е рекурсия?

Повторение чрез позоваване на себе си

Рекурсивна функция: дефинира се чрез себе си

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

- 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”?

## Рекурсивни уравнения

$$y = y^2 - 5$$

$$y = f(y) \quad F = f(F)$$

Какво означава “да дефинираме функция чрез себе си”?

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

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

# 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  $\Gamma$  е изчислим оператор, то уравнението  $F = \Gamma(F)$  има единствено най-малко решение  $f$

# Най-малка неподвижна точка

## Теорема (Knaster-Tarski)

Ако  $\Gamma$  е изчислим оператор, то уравнението  $F = \Gamma(F)$  има единствено най-малко решение  $f$  (най-малка неподвижна точка на  $\Gamma$ ).

# Най-малка неподвижна точка

## Теорема (Knaster-Tarski)

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

$$\begin{aligned} & \{ (1,2) \} \xrightarrow{\Gamma} \{ \{ \} \} \\ & \quad \vdots \\ & \quad \{ (1,3), (2,4) \} \\ & \quad \vdots \\ & \quad \{ (1,3) \} \end{aligned}$$

# Най-малка неподвижна точка

## Теорема (Knaster-Tarski)

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

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

# Най-малка неподвижна точка

## Теорема (Knaster-Tarski)

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

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

 $N \rightarrow N$ 

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

~~$f(x) = 1 + f(x - 1)$~~   $\forall x$

# Най-малка неподвижна точка

## Теорема (Knaster-Tarski)

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

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

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

```
(define (f x) (+ 1 (f (- x 1))))
(f 0) —→ ?
```

# Най-малка неподвижна точка

## Теорема (Knaster-Tarski)

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

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

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

```
(define (f x) (+ 1 (f (- x 1))))
(f 0) —> ?
```

$f$  е “празната функция”, т.е.  $\text{dom}(f) = \emptyset$ .

# Операционна и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

## Операционна и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

Нека  $(\text{define } (f \ x) \ \Gamma[f])$  е рекурсивно дефинирана функция.

## Операционна и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

Нека  $(\text{define } (f \ x) \ \Gamma[f])$  е рекурсивно дефинирана функция.

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

## Операционна и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

Нека  $(\text{define } (f \ x) \ \Gamma[f])$  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 и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

Нека  $(\text{define } (f \ x) \ \Gamma[f])$  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

$(f \ a) \rightarrow \Gamma[f] [x \mapsto a] \rightarrow \dots$

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

# Операционна и денотационна семантика

Два подхода за описание на смисъла на функциите в Scheme.

Нека  $(\text{define } (f \ x) \ \Gamma[f])$  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

$(f \ a) \rightarrow \Gamma[f] [x \mapsto a] \rightarrow \dots$

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

$$\Gamma(F) = F \quad \text{Daut: } \Gamma(f) \supseteq f$$

$$f_0 := \emptyset$$

$$f_1 := T(f_0)$$

$$f_2 := T(f_1) \quad \dots$$

$$\vdots$$
$$f_{n+1} := T(f_n)$$

$$\text{Ab } f_n = f_{n+1}$$

$$f_0 \not\subseteq f_1 \not\subseteq f_2 \not\subseteq \dots \not\subseteq f_n \not\subseteq \dots$$

$$T(F)(n) := \begin{cases} 1 & , n=0 \\ n \cdot f(n-1) & , n > 0 \end{cases}$$

$$f_0 := \emptyset$$

$$f_1 := T(\emptyset) \quad f_1(n) = \begin{cases} 1 & , n=0 \\ \cancel{n \cdot \emptyset(n-1)} & , n > 0 \\ \cancel{n!} & \end{cases}$$

$$f_2 := T(f_1) \quad f_2(n) = \begin{cases} 1 & , n=0 \\ \cancel{n \cdot f_1(n-1)} & , n > 0 \\ 1. f_1(0)=1 & \quad n=1 \\ \text{neg. def.} & \quad n > 1 \end{cases}$$

$$T(f)(n) := \begin{cases} 1 & , n=0 \\ n \cdot f(n-1) & , n > 0 \end{cases}$$

$$f_k(n) := \begin{cases} n! & , n < k \\ 1! & , n \geq k \end{cases}$$

$$\sqrt[n]{f} := \lim_{k \rightarrow \infty} f_k = \bigcup_{k \rightarrow \infty} f_k = n!$$

# Оценка на рекурсивна функция

(fact 4)

# Оценка на рекурсивна функция

```
(fact 4)
      ↓
(if (= 4 0) 1 (* 4 (fact (- 4 1))))
```

# Оценка на рекурсивна функция

```
(fact 4)
      ↓
(* 4 (fact 3))
```

# Оценка на рекурсивна функция

```
(fact 4)
      ↓
(* 4 (fact 3))
      ↓
(* 4 (if (= 3 0) 1 (* 3 (fact (- 3 1))))))
```

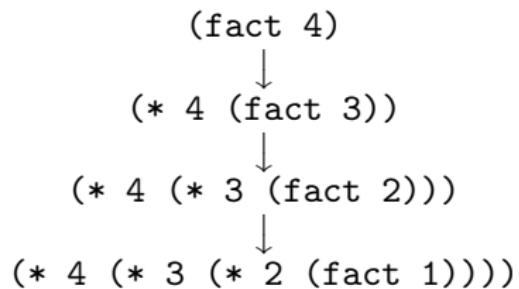
# Оценка на рекурсивна функция

```
(fact 4)
      ↓
(* 4 (fact 3))
      ↓
(* 4 (* 3 (fact 2)))
```

# Оценка на рекурсивна функция

```
(fact 4)
      ↓
(* 4 (fact 3))
      ↓
(* 4 (* 3 (fact 2)))
      ↓
(* 4 (* 3 (if (= 2 0) 1 (* 2 (fact (- 2 1)))))))
```

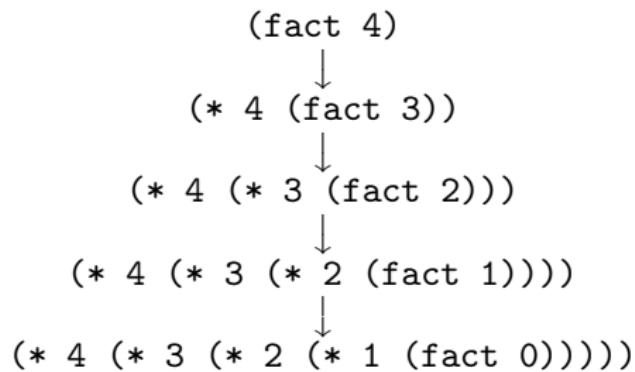
# Оценка на рекурсивна функция



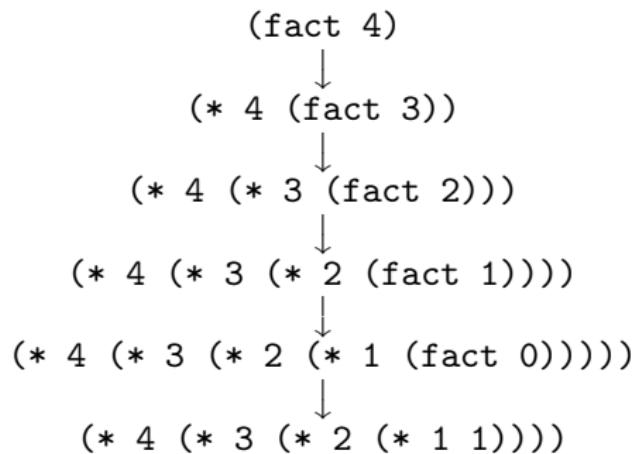
# Оценка на рекурсивна функция

```
(fact 4)
      ↓
(* 4 (fact 3))
      ↓
(* 4 (* 3 (fact 2)))
      ↓
(* 4 (* 3 (* 2 (fact 1))))
      ↓
(* 4 (* 3 (* 2 (if (= 1 0) 1 (* 1 (fact (- 1 1))))))))
```

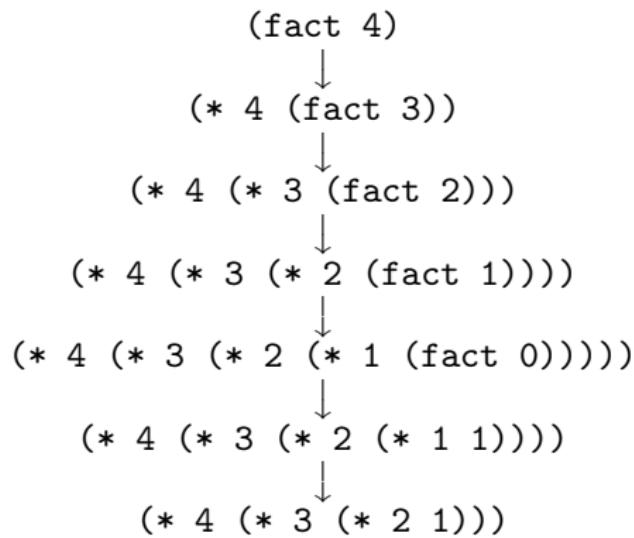
# Оценка на рекурсивна функция



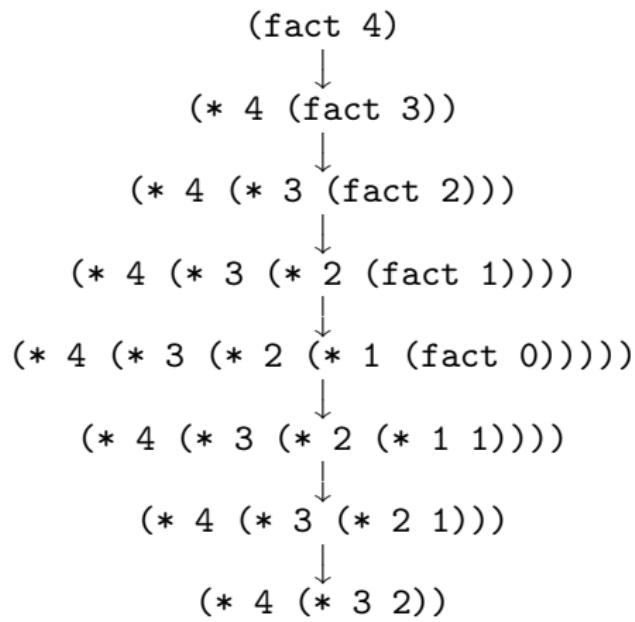
# Оценка на рекурсивна функция



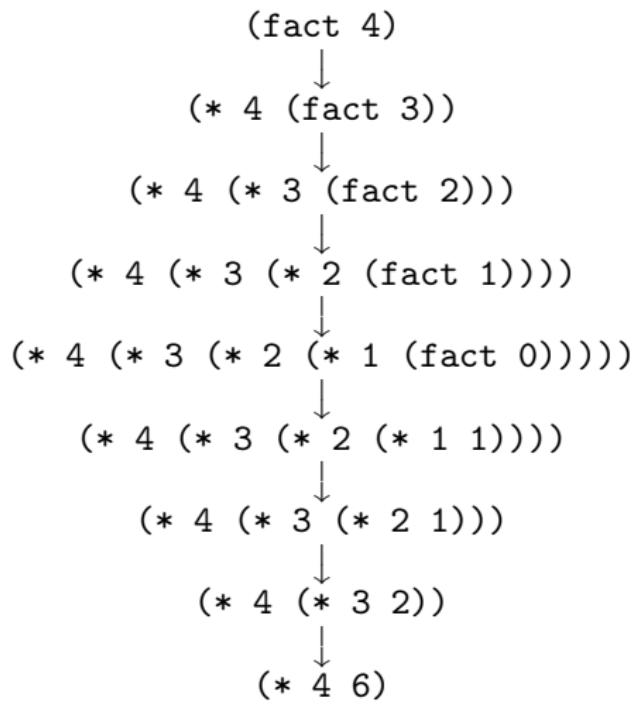
# Оценка на рекурсивна функция



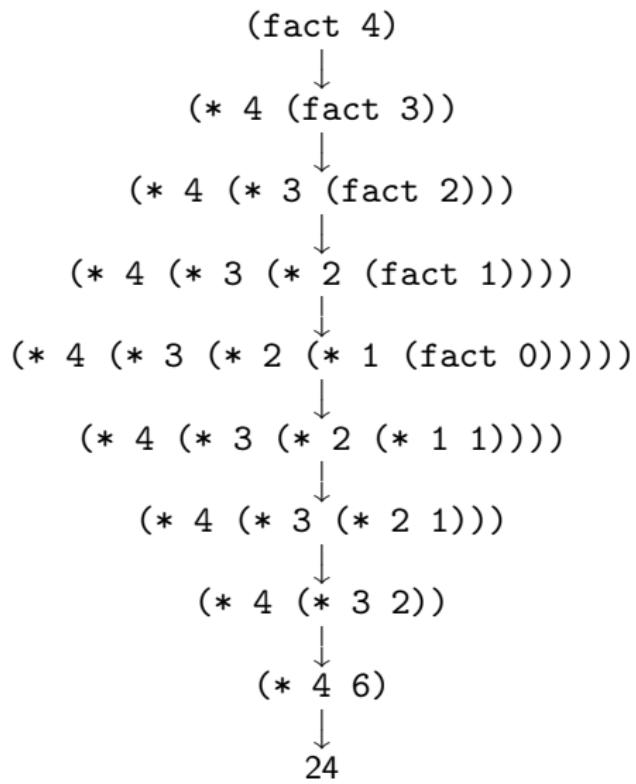
# Оценка на рекурсивна функция



# Оценка на рекурсивна функция



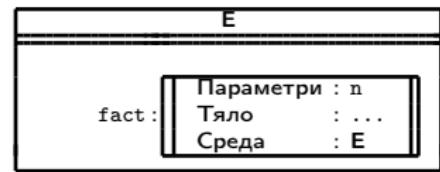
# Оценка на рекурсивна функция



# Оценка на рекурсивна функция в среда

{E}

(fact 4)



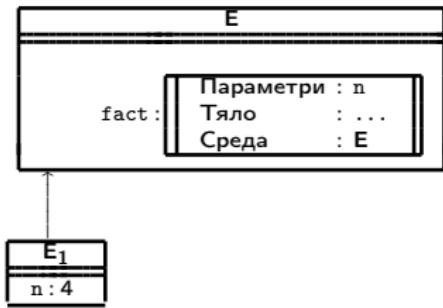
# Оценка на рекурсивна функция в среда

{E}

(fact 4)

{E<sub>1</sub>}

(if (= n 0) 1 (\* n (fact (- n 1))))

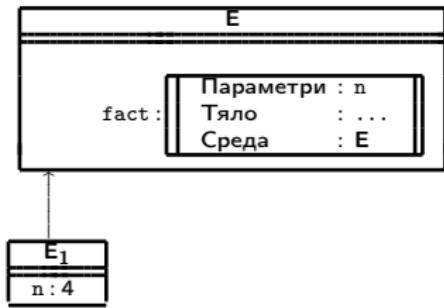


# Оценка на рекурсивна функция в среда

{E}

{E<sub>1</sub>}

(fact 4)  
 ↓  
 (\* 4 (fact 3))

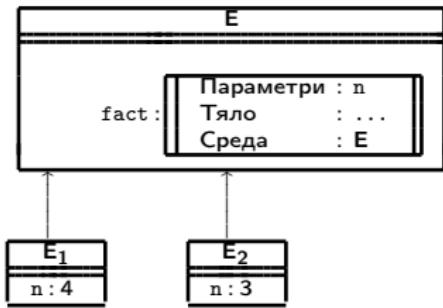


# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
             ↓
{E1}         (* 4 (fact 3))
             ↓
{E2}     (* 4 (if (= n 0) 1 (* n (fact (- n 1)))))

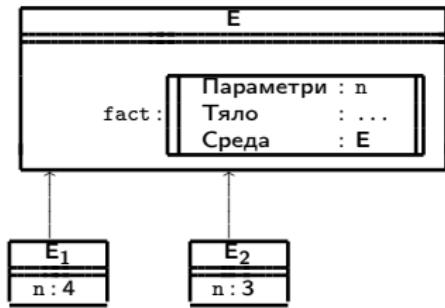

```



# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
             ↓
{E1}         (* 4 (fact 3))
             ↓
{E2}         (* 4 (* 3 (fact 2)))
  
```

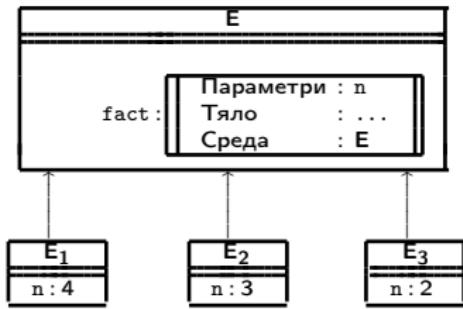


# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
             ↓
{E1}     (* 4 (fact 3))
             ↓
{E2}     (* 4 (* 3 (fact 2)))
             ↓
{E3}     (* 4 (* 3 (if (= n 0) 1 (* n (fact (- n 1)))))))

```



# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
             ↓
{E1}     (* 4 (fact 3))
             ↓
{E2}     (* 4 (* 3 (fact 2)))
             ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))

```

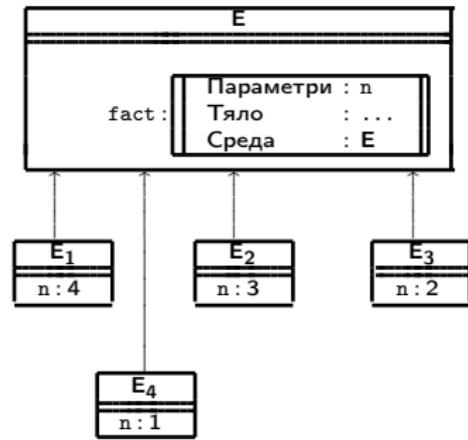


# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
              ↓
{E1}     (* 4 (fact 3))
              ↓
{E2}     (* 4 (* 3 (fact 2)))
              ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
              ↓
{E4}     (* 4 (* 3 (* 2 (if (= n 0) 1 (* n (fact (- n 1)))))))

```

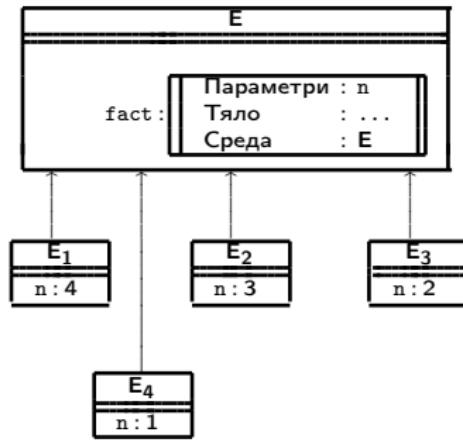


# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
              ↓
{E1}     (* 4 (fact 3))
              ↓
{E2}     (* 4 (* 3 (fact 2)))
              ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 (fact 0)))))

    
```

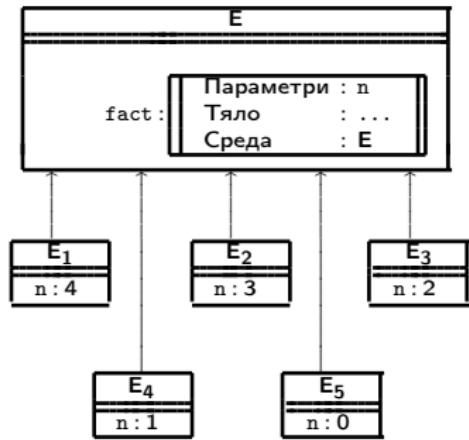


# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
             ↓
{E1}     (* 4 (fact 3))
             ↓
{E2}     (* 4 (* 3 (fact 2)))
             ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
             ↓
{E4}     (* 4 (* 3 (* 2 (* 1 (fact 0)))))
             ↓
{E5}     (* 4 (* 3 (* 2 (* 1 (if (= n 0) 1 (* n (fact (- n 1))))))))

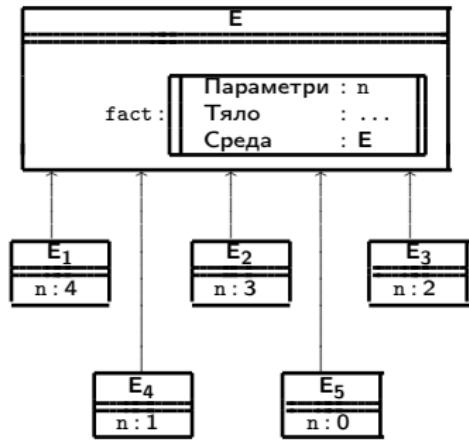
```



# Оценка на рекурсивна функция в среда

```

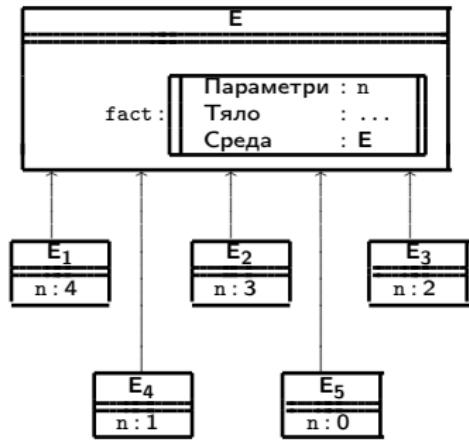
{E}           (fact 4)
              ↓
{E1}     (* 4 (fact 3))
              ↓
{E2}     (* 4 (* 3 (fact 2)))
              ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 (fact 0)))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 1))))
  
```



# Оценка на рекурсивна функция в среда

```

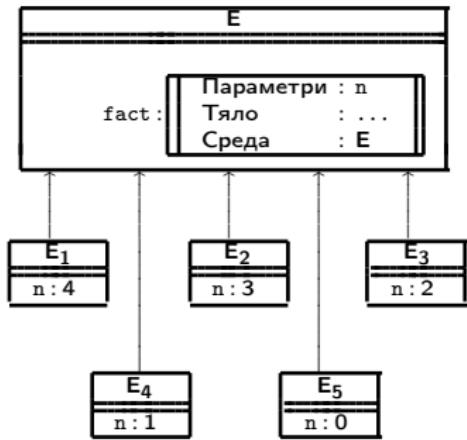
{E}           (fact 4)
              ↓
{E1}     (* 4 (fact 3))
              ↓
{E2}     (* 4 (* 3 (fact 2)))
              ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 (fact 0)))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 1))))
              ↓
{E3}     (* 4 (* 3 (* 2 1)))
  
```



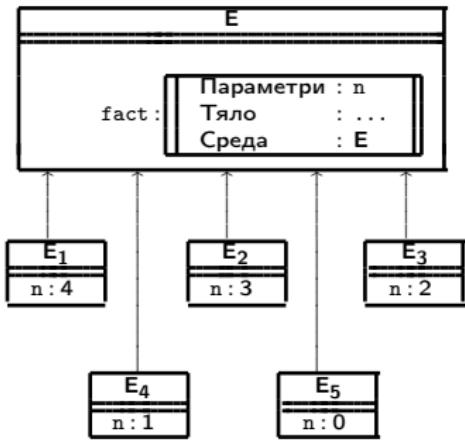
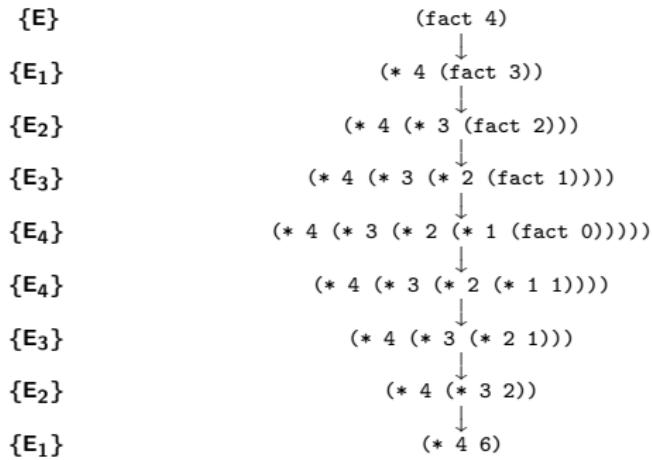
# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
              ↓
{E1}     (* 4 (fact 3))
              ↓
{E2}     (* 4 (* 3 (fact 2)))
              ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 (fact 0)))))
              ↓
{E4}     (* 4 (* 3 (* 2 (* 1 1))))
              ↓
{E3}     (* 4 (* 3 (* 2 1)))
              ↓
{E2}     (* 4 (* 3 2))
  
```



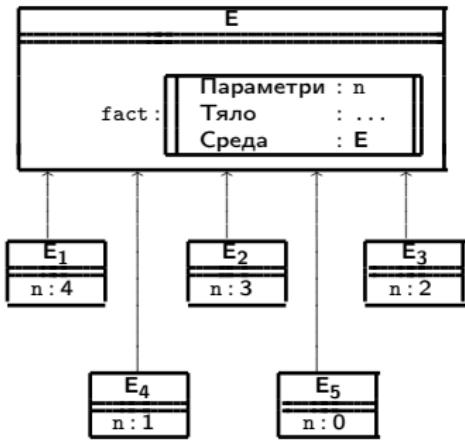
# Оценка на рекурсивна функция в среда



# Оценка на рекурсивна функция в среда

```

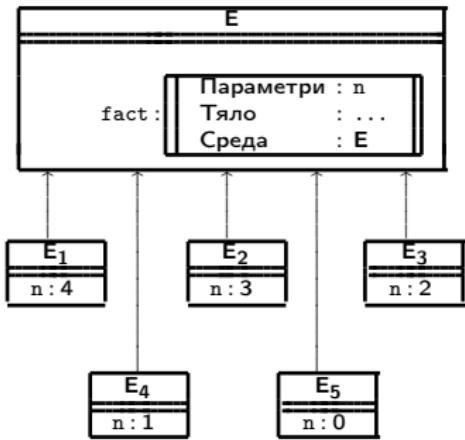
{E}
      (fact 4)
      ↓
(* 4 (fact 3))
      ↓
(* 4 (* 3 (fact 2)))
      ↓
(* 4 (* 3 (* 2 (fact 1))))
      ↓
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
      ↓
(* 4 (* 3 (* 2 (* 1 1))))
      ↓
(* 4 (* 3 (* 2 1)))
      ↓
(* 4 (* 3 2))
      ↓
(* 4 6)
      ↓
24
  
```



# Оценка на рекурсивна функция в среда

```

{E}           (fact 4)
             ↓
{E1}     (* 4 (fact 3))
             ↓
{E2}     (* 4 (* 3 (fact 2)))
             ↓
{E3}     (* 4 (* 3 (* 2 (fact 1))))
             ↓
{E4}     (* 4 (* 3 (* 2 (* 1 (fact 0)))))
             ↓
{E4}     (* 4 (* 3 (* 2 (* 1 1))))
             ↓
{E3}     (* 4 (* 3 (* 2 1)))
             ↓
{E2}     (* 4 (* 3 2))
             ↓
{E1}     (* 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;  
}
```

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

# 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))
```

# 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))
```

# 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))
```

# 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))
```

# 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))
```

# 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))
```

# 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

(fact 4)

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
```

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(if (<= 1 4) (for 4 (* 1 1) (+ 1 1)) 1)
```

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(for 4 1 2)
```

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(for 4 1 2)
  ↓
(if (<= 2 4) (for 4 (* 1 2) (+ 2 1)) 2)
```

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(for 4 1 2)
  ↓
(for 4 2 3)
```

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(for 4 1 2)
  ↓
(for 4 2 3)
  ↓
(if (<= 3 4) (for 4 (* 2 3) (+ 3 1)) 6)
```

# Оценка на итеративен факториел

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

# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(for 4 1 2)
  ↓
(for 4 2 3)
  ↓
(for 4 6 4)
  ↓
(if (<= 4 4) (for 4 (* 6 4) (+ 4 1)) 24)
```

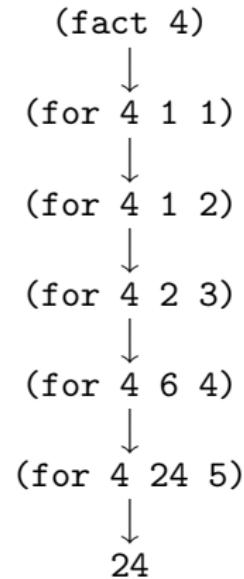
# Оценка на итеративен факториел

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

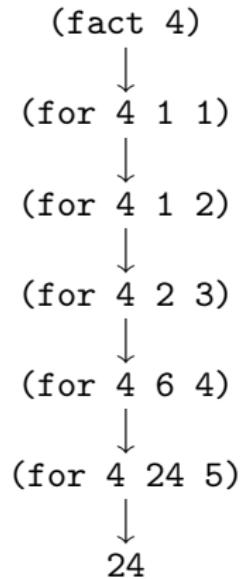
# Оценка на итеративен факториел

```
(fact 4)
  ↓
(for 4 1 1)
  ↓
(for 4 1 2)
  ↓
(for 4 2 3)
  ↓
(for 4 6 4)
  ↓
(for 4 24 5)
  ↓
(if (<= 5 4) (for 4 (* 24 5) (+ 5 1)) 24)
```

# Оценка на итеративен факториел



# Оценка на итеративен факториел



Линеен итеративен процес

# Оценка на итеративен факториел със среди

{E}

(fact 4)

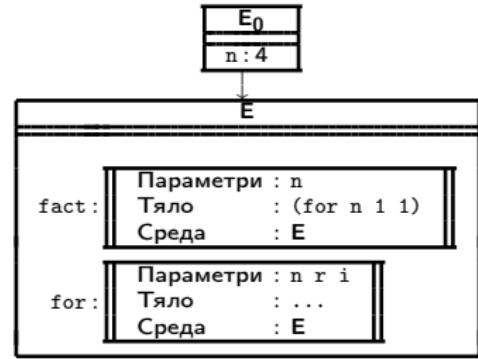


# Оценка на итеративен факториел със среди

{E}

{E<sub>0</sub>}

(fact 4)  
 ↓  
 (for n 1 1)

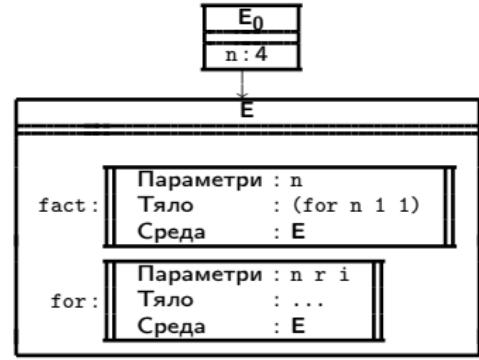


# Оценка на итеративен факториел със среди

{E}

{E<sub>0</sub>}

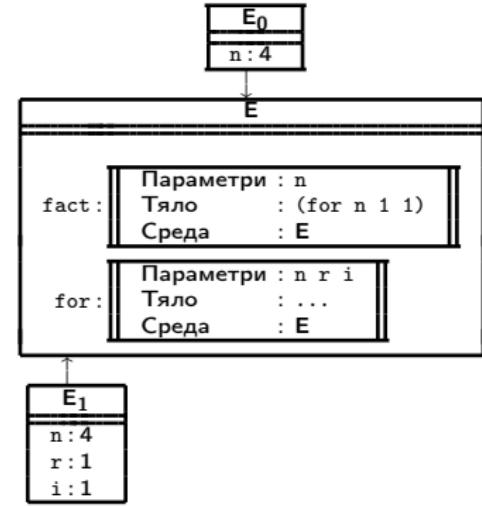
(fact 4)  
 ↓  
 (for 4 1 1)



# Оценка на итеративен факториел със среди

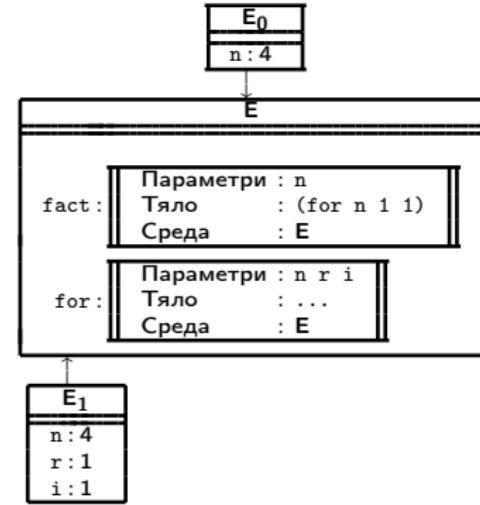
```

{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}  (if (<= i n) (for n (* r i) (+ i 1)) r)
  
```



# Оценка на итеративен факториел със среди

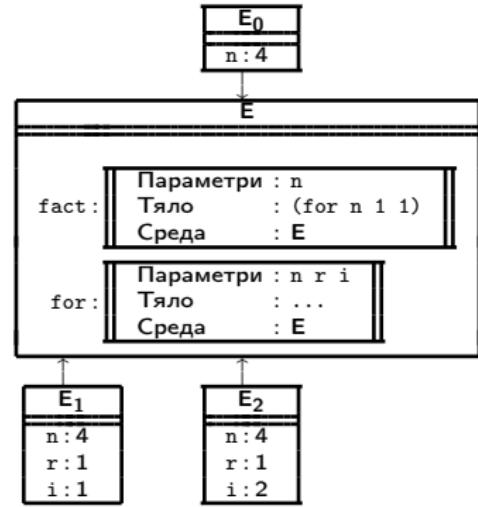
{E}	(fact 4)
{E <sub>0</sub> }	↓
{E <sub>1</sub> }	(for 4 1 1)
	↓
	(for 4 1 2)



# Оценка на итеративен факториел със среди

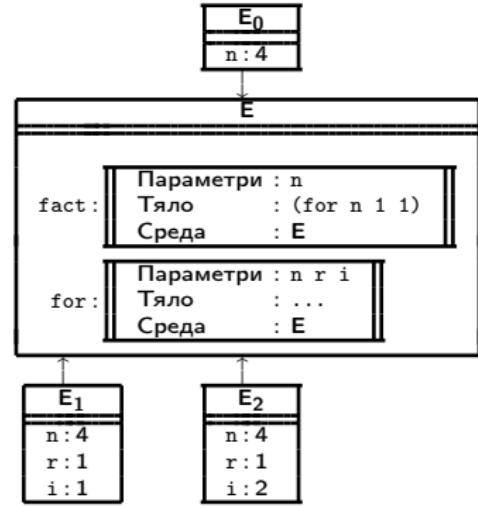
```

{E}           (fact 4)
              ↓
{E0}      (for 4 1 1)
              ↓
{E1}      (for 4 1 2)
              ↓
{E2}      (if (<= i n) (for n (* r i) (+ i 1)) r)
  
```



# Оценка на итеративен факториел със среди

{E}	(fact 4)
{E <sub>0</sub> }	↓
	(for 4 1 1)
{E <sub>1</sub> }	↓
	(for 4 1 2)
{E <sub>2</sub> }	↓
	(for 4 2 3)



# Оценка на итеративен факториел със среди

```

{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}          (for 4 1 2)
              ↓
{E2}          (for 4 2 3)
              ↓
{E3}  (if (<= i n) (for n (* r i) (+ i 1)) r)
    
```



# Оценка на итеративен факториел със среди

```

{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}          (for 4 1 2)
              ↓
{E2}          (for 4 2 3)
              ↓
{E3}          (for 4 6 4)
  
```



# Оценка на итеративен факториел със среди

```

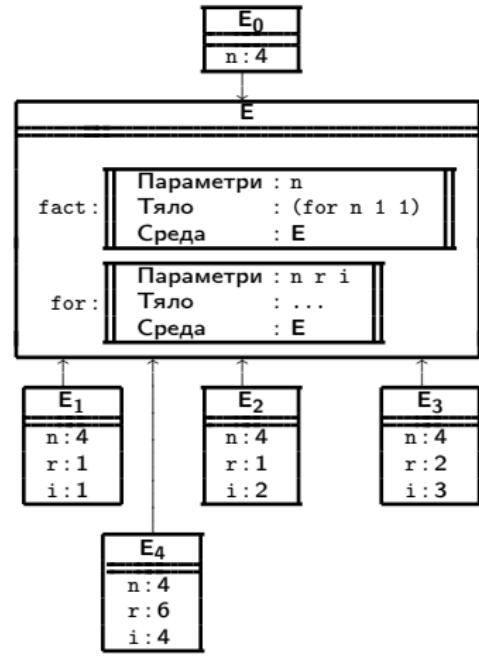
{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}          (for 4 1 2)
              ↓
{E2}          (for 4 2 3)
              ↓
{E3}          (for 4 6 4)
              ↓
{E4}  (if (<= i n) (for n (* r i) (+ i 1)) r)
    
```



# Оценка на итеративен факториел със среди

```

{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}          (for 4 1 2)
              ↓
{E2}          (for 4 2 3)
              ↓
{E3}          (for 4 6 4)
              ↓
{E4}          (for 4 24 5)
  
```



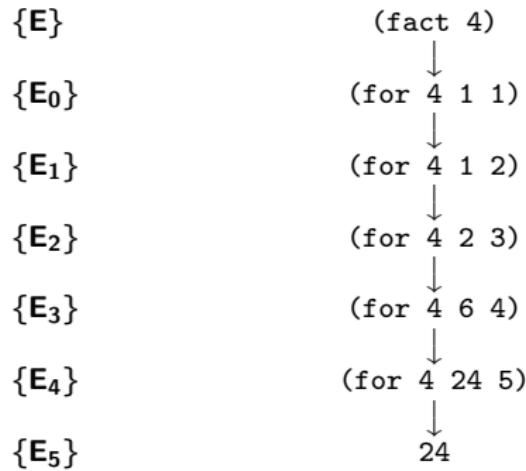
# Оценка на итеративен факториел със среди

```

{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}          (for 4 1 2)
              ↓
{E2}          (for 4 2 3)
              ↓
{E3}          (for 4 6 4)
              ↓
{E4}          (for 4 24 5)
              ↓
{E5}  (if (<= i n) (for n (* r i) (+ i 1)) r)
    
```



# Оценка на итеративен факториел със среди



# 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))

```

# Рекурсивен и итеративен процес

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

(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
```

```
(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))
```

# Опашкова рекурсия

- Функциите, в които има отложени операции генерираят същински рекурсивни процеси

# Опашкова рекурсия

- Функциите, в които има отложени операции генерираят същински рекурсивни процеси
- Рекурсивно извикване, при което няма отложена операция се нарича **опашкова рекурсия**

# 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**

# 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

# 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

# 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

# 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))

```

# Оценка на итеративен факториел със среди

```

{E}           (fact 4)
              ↓
{E0}          (for 4 1 1)
              ↓
{E1}          (for 4 1 2)
              ↓
{E2}          (for 4 2 3)
              ↓
{E3}          (for 4 6 4)
              ↓
{E4}          (for 4 24 5)
              ↓
{E5}           24
  
```



# Вложени дефиниции

- (define (<функция> {<параметър>}) {<дефиниция>}) <тяло>

## Вложени дефиниции

- `(define (<функция> {<параметър>}) {<дефиниция>}) <тяло>`
- При извикване на `<функция>` първо се оценяват всички `<дефиниция>` и след това се оценява `<тяло>`

## Вложени дефиниции

- (define (<функция> {<параметър>}) {<дефиниция>}) <тяло>)
- При извикване на <функция> първо се оценяват всички <дефиниция> и след това се оценява <тяло>
- Вложените дефиниции се оценяват и записват в средата, която се **оценява** функцията, а не в средата, в която е **дефинирана**

## Вложени дефиниции

- (define (<функция> {<параметър>}) {<дефиниция>}) <тяло>)
- При извикване на <функция> първо се оценяват всички <дефиниция> и след това се оценява <тяло>
- Вложените дефиниции се оценяват и записват в средата, която се оценява функцията, а не в средата, в която е дефинирана
- **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	
dist :	Параметри : x1 y1 x2 y2
	Тяло : ...
	Среда : E

# Оценка на вложени функции

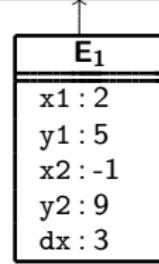
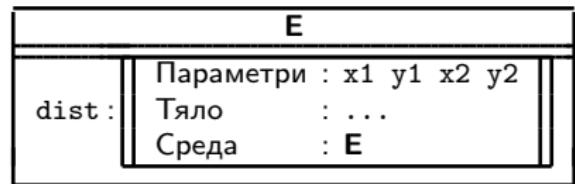
{E}

(dist 2 5 -1 9)



# Оценка на вложени функции

{E}            (dist 2 5 -1 9)  
               ↓  
 {E1}        (define dx (- x2 x1))



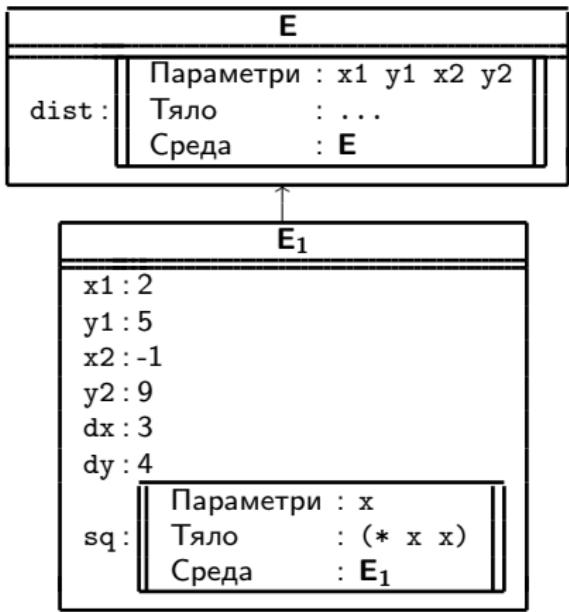
# Оценка на вложени функции

```
{E}      (dist 2 5 -1 9)
          ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
```



# Оценка на вложени функции

```
{E}      (dist 2 5 -1 9)
          ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
```



# Оценка на вложени функции

```
{E}      (dist 2 5 -1 9)
          ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
{E1}    (sqrt (+ (sq dx) (sq dy)))
```



# Оценка на вложени функции

```
{E}      (dist 2 5 -1 9)
          ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
{E1}    (sqrt (+ (sq dx) (sq dy)))
          ↓
{E2}    (sqrt (+ (* x x) (sq dy)))
```



# Оценка на вложени функции

```

{E}      (dist 2 5 -1 9)
         ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
{E1}    (sqrt (+ (sq dx) (sq dy)))
         ↓
{E2}    (sqrt (+ (* x x) (sq dy)))
         ↓
{E3}    (sqrt (+ 9 (* x x)))
  
```



# Оценка на вложени функции

```

{E}      (dist 2 5 -1 9)
         ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
{E1}    (sqrt (+ (sq dx) (sq dy)))
         ↓
{E2}    (sqrt (+ (* x x) (sq dy)))
         ↓
{E3}    (sqrt (+ 9 (* x x)))
         ↓
{E1}    (sqrt (+ 9 16))
  
```



# Оценка на вложени функции

```

{E}      (dist 2 5 -1 9)
         ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
{E1}    (sqrt (+ (sq dx) (sq dy)))
         ↓
{E2}    (sqrt (+ (* x x) (sq dy)))
         ↓
{E3}    (sqrt (+ 9 (* x x)))
         ↓
{E1}    (sqrt (+ 9 16))
         ↓
{E1}    (sqrt 25)
  
```



# Оценка на вложени функции

```

{E}      (dist 2 5 -1 9)
         ↓
{E1}    (define dx (- x2 x1))
{E1}    (define dy (- y2 y1))
{E1}    (define (sq x) (* x x))
{E1}    (sqrt (+ (sq dx) (sq dy)))
         ↓
{E2}    (sqrt (+ (* x x) (sq dy)))
         ↓
{E3}    (sqrt (+ 9 (* x x)))
         ↓
{E1}    (sqrt (+ 9 16))
         ↓
{E1}    (sqrt 25)
         ↓
{E1}    5
  
```



## Вложена помощна итеративна функция

При итеративни функции е удобно помощната функция да е вложена.

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

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

## Вложена помощна итеративна функция

При итеративни функции 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))
```

## Вложена помощна итеративна функция

При итеративни функции 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.