

# 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**.
- 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`  $\longrightarrow$  **Грешка!**
- `(define r 5)`
- `(+ r 3)`  $\longrightarrow$  8
- `(define (f x) (* x r))`
- `(f 3)`  $\longrightarrow$  15
- `(f r)`  $\longrightarrow$  25

E							
a :	8						
r :	5						
f :	<table border="1"> <tr> <td>Параметри :</td><td>x</td></tr> <tr> <td>Тяло :</td><td>(* x r)</td></tr> <tr> <td>Среда :</td><td>E</td></tr> </table>	Параметри :	x	Тяло :	(* x r)	Среда :	E
Параметри :	x						
Тяло :	(* x r)						
Среда :	E						

# Функции и среди

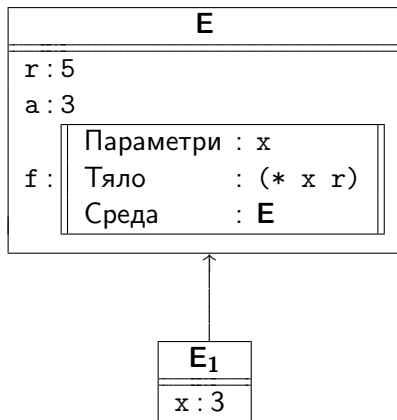
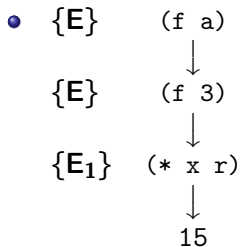
- Всяка функция  $f$  пази указател към средата  $E$ , в която е дефинирана.
- При извикване на  $f$ :
  - създава се нова среда  $E_1$ , която разширява  $E$
  - в  $E_1$  всеки символ означаващ формален параметър се свързва с оценката на фактическия параметър
  - тялото на  $f$  се оценява в  $E_1$

## 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))`



# Какво е рекурсия?



# Какво е рекурсия?





# Какво е рекурсия?

Повторение чрез позоваване на себе си

Рекурсивна функция: дефинира се чрез себе си

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

- 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  $\Gamma$  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 пресмятаща  $f$  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)  $\longrightarrow$  ?
```

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

# 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

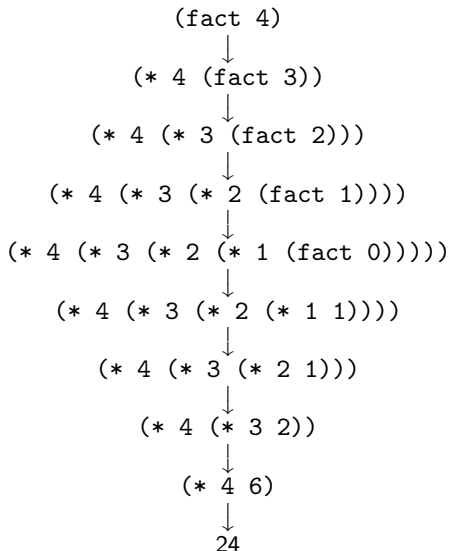
Разглеждаме редицата от последователни оценки на комбинации  
 $(f\ a) \rightarrow \Gamma[f]\ [x \mapsto a] \rightarrow \dots$

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

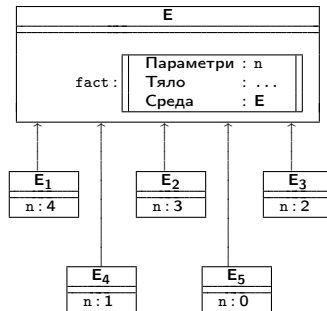
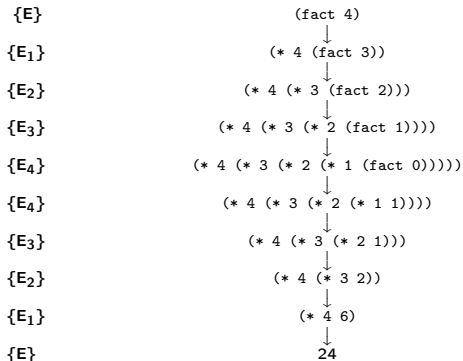
Функциите в Scheme имат дуален, но еквивалентен смисъл:

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

# Оценка на рекурсивна функция



# Оценка на рекурсивна функция в среда



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))
```

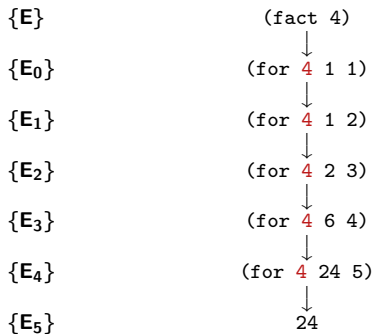
# Оценка на итеративен факториел

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

Линеен итеративен процес



# 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))
  
```

```

(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))

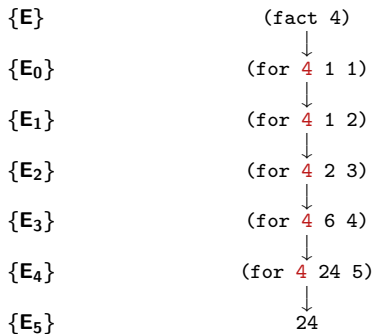
```

```

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

```

# Оценка на итеративен факториел със среди



# 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)
      ↓
{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 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.