Списъци

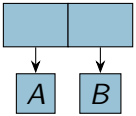
Трифон Трифонов

Функционално програмиране, 2023/24 г.

25 октомври–15 ноември 2023 г.

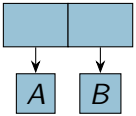
Наредени двойки

```
(A . B)
```



Наредени двойки

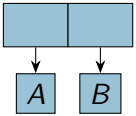
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`

Наредени двойки

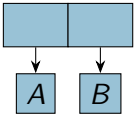
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>

Наредени двойки

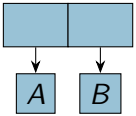
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>
- `(car` <израз>`)`

Наредени двойки

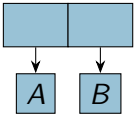
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>
- `(car` <израз>`)`
- **Първият** компонент на двойката, която е оценката на <израз>

Наредени двойки

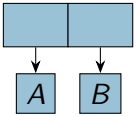
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>
- `(car` <израз>`)`
- **Първият** компонент на двойката, която е оценката на <израз>
- `(cdr` <израз>`)`

Наредени двойки

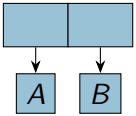
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>
- `(car` <израз>`)`
- **Първият** компонент на двойката, която е оценката на <израз>
- `(cdr` <израз>`)`
- **Вторият** компонент на двойката, която е оценката на <израз>

Наредени двойки

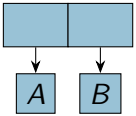
```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>
- `(car` <израз>`)`
- **Първият** компонент на двойката, която е оценката на <израз>
- `(cdr` <израз>`)`
- **Вторият** компонент на двойката, която е оценката на <израз>
- `(pair?` <израз>`)`

Наредени двойки

```
(A . B)
```



- `(cons` <израз$_1$> <израз$_2$>`)`
- Наредена двойка от оценките на <израз$_1$> и <израз$_2$>
- `(car` <израз>`)`
- **Първият** компонент на двойката, която е оценката на <израз>
- `(cdr` <израз>`)`
- **Вторият** компонент на двойката, която е оценката на <израз>
- `(pair?` <израз>`)`
- Проверява дали оценката на <израз> е наредена двойка

Примери

```
(cons (cons 2 3) (cons 8 13))
```



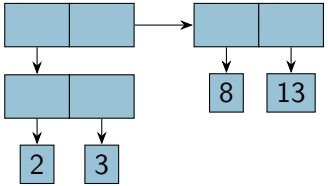
```
((2 . 3) . (8 . 13))
```

Примери

```
(cons (cons 2 3) (cons 8 13))
```

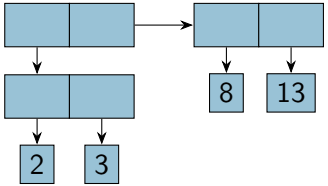
↓

```
((2 . 3) . (8 . 13))
```



Примери

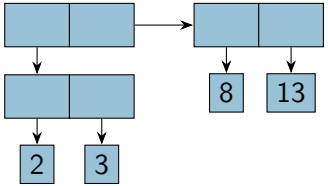
```
(cons (cons 2 3) (cons 8 13))
              ↓
     ((2 . 3) . (8 . 13))
```



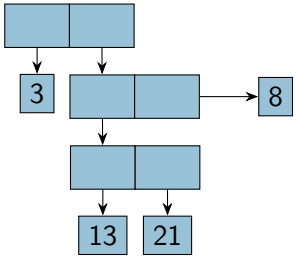

```
(cons 3 (cons (cons 13 21) 8))
              ↓
     (3 . ((13 . 21) . 8))
```

Примери

```
(cons (cons 2 3) (cons 8 13))
              ↓
     ((2 . 3) . (8 . 13))
```




```
(cons 3 (cons (cons 13 21) 8))
              ↓
     (3 . ((13 . 21) . 8))
```



S-изрази

Дефиниция

S-израз наричаме:

- атоми (булеви, числа, знаци, символи, низове, функции)
- наредени двойки $(S_1 \ . \ S_2)$, където S_1 и S_2 са S-изрази

S-изрази

Дефиниция

S-израз наричаме:

- атоми (булеви, числа, знаци, символи, низове, функции)
- наредени двойки $(S_1 \ . \ S_2)$, където S_1 и S_2 са S-изрази

S-изразите са най-общият тип данни в Scheme.

С тяхна помощ могат да се дефинират произволно сложни структури от данни.

All you need is λ — наредени двойки

Можем да симулираме `cons`, `car` и `cdr` чрез `lambda`!

All you need is λ — наредени двойки

Можем да симулираме `cons`, `car` и `cdr` чрез `lambda`!

Вариант №1:

```
(define (lcons x y) (lambda (p) (if p x y)))
(define (lcar z) (z #t))
(define (lcdr z) (z #f))
```

All you need is λ — наредени двойки

Можем да симулираме `cons`, `car` и `cdr` чрез `lambda`!

Вариант №1:

```
(define (lcons x y) (lambda (p) (if p x y)))
(define (lcar z) (z #t))
(define (lcdr z) (z #f))
```

Вариант №2:

```
(define (lcons x y) (lambda (p) (p x y)))
(define (lcar z) (z (lambda (x y) x)))
(define (lcdr z) (z (lambda (x y) y)))
```

Списъци в Scheme

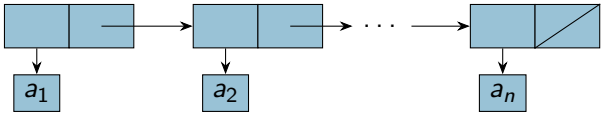
Дефиниция

1. Празният списък `()` е списък
2. `(h . t)` е списък ако `t` е списък
   - `h` — глава на списъка
   - `t` — опашка на списъка

Списъци в Scheme

Дефиниция

1. Празният списък `()` е списък
2. `(h . t)` е списък ако `t` е списък
   - `h` — глава на списъка
   - `t` — опашка на списъка



$(a_1 \ . \ (a_2 \ . \ (\ \ldots (\ a_n \ . \ () \ ) \ ) \ ) \ ) \iff (a_1 \ a_2 \ \ldots \ a_n)$

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`
- `(list?` <израз>`)` — дали <израз> е списък

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`
- `(list?` <израз>`)` — дали <израз> е списък
  - `(define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))`

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`
- `(list?` <израз>`)` — дали <израз> е списък
  - `(define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))`
- `(list` {<израз>}`)` — построява списък с елементи <израз>

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`
- `(list?` <израз>`)` — дали <израз> е списък
  - `(define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))`
- `(list` {<израз>}`)` — построява списък с елементи <израз>
- `(list` <израз$_1$> <израз$_2$> ... <израз$_n$>`)` $\Longleftrightarrow$
  `(cons` <израз$_1$> `(cons` <израз$_2$> ... `(cons` <израз$_n$> `'()))))`

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`
- `(list?` <израз>`)` — дали <израз> е списък
  - `(define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))`
- `(list` {<израз>}`)` — построява списък с елементи <израз>
- `(list` <израз$_1$> <израз$_2$> ... <израз$_n$>`)` $\iff$
  `(cons` <израз$_1$> `(cons` <израз$_2$> ... `(cons` <израз$_n$> `'()))))`
- `(cons` <глава> <опашка>`)` — списък с <глава> и <опашка>

Вградени функции за списъци

- (null? <израз>) — дали <израз> е празният списък ()
- (list? <израз>) — дали <израз> е списък
  - (define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))
- (list {<израз>}) — построява списък с елементи <израз>
- (list <израз$_1$> <израз$_2$> ... <израз$_n$>) $\iff$
  (cons <израз$_1$> (cons <израз$_2$> ... (cons <израз$_n$> '()))))
- (cons <глава> <опашка>) — списък с <глава> и <опашка>
- (car <списък>) — главата на <списък>

Вградени функции за списъци

- (null? <израз>) — дали <израз> е празният списък ()
- (list? <израз>) — дали <израз> е списък
  - (define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))
- (list {<израз>}) — построява списък с елементи <израз>
- (list <израз$_1$> <израз$_2$> ... <израз$_n$>) $\iff$
  (cons <израз$_1$> (cons <израз$_2$> ... (cons <израз$_n$> '()))))
- (cons <глава> <опашка>) — списък с <глава> и <опашка>
- (car <списък>) — главата на <списък>
- (cdr <списък>) — опашката на <списък>

Вградени функции за списъци

- (null? <израз>) — дали <израз> е празният списък ()
- (list? <израз>) — дали <израз> е списък
  - (define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))
- (list {<израз>}) — построява списък с елементи <израз>
- (list <израз$_1$> <израз$_2$> ... <израз$_n$>) $\iff$
  (cons <израз$_1$> (cons <израз$_2$> ... (cons <израз$_n$> '()))))
- (cons <глава> <опашка>) — списък с <глава> и <опашка>
- (car <списък>) — главата на <списък>
- (cdr <списък>) — опашката на <списък>
- () не е наредена двойка!

Вградени функции за списъци

- `(null?` <израз>`)` — дали <израз> е празният списък `()`
- `(list?` <израз>`)` — дали <израз> е списък
  - `(define (list? l) (or (null? l) (and (pair? l) (list? (cdr l)))))`
- `(list` {<израз>}`)` — построява списък с елементи <израз>
- `(list` <израз$_1$> <израз$_2$> ... <израз$_n$>`)` $\Longleftrightarrow$
  `(cons` <израз$_1$> `(cons` <израз$_2$> ... `(cons` <израз$_n$> `'())))))`
- `(cons` <глава> <опашка>`)` — списък с <глава> и <опашка>
- `(car` <списък>`)` — главата на <списък>
- `(cdr` <списък>`)` — опашката на <списък>
- `()` не е наредена двойка!
- `(car '())` $\longrightarrow$ Грешка!, `(cdr '())` $\longrightarrow$ Грешка!

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow\ ?\ \longleftarrow$ `(cadr l)`

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow a_2 \longleftarrow$ `(cadr l)`

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow a_2 \longleftarrow$ `(cadr l)`
- `(cdr (cdr l))` $\longrightarrow\ ?\ \longleftarrow$ `(cddr l)`

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow a_2 \longleftarrow$ `(cadr l)`
- `(cdr (cdr l))` $\longrightarrow (a_3 \ldots a_n) \longleftarrow$ `(cddr l)`

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow a_2 \longleftarrow$ `(cadr l)`
- `(cdr (cdr l))` $\longrightarrow (a_3 \ldots a_n) \longleftarrow$ `(cddr l)`
- `(car (cdr (cdr l)))` $\longrightarrow ? \longleftarrow$ `(caddr l)`

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow a_2 \longleftarrow$ `(cadr l)`
- `(cdr (cdr l))` $\longrightarrow (a_3 \ldots a_n) \longleftarrow$ `(cddr l)`
- `(car (cdr (cdr l)))` $\longrightarrow a_3 \longleftarrow$ `(caddr l)`

Съкратени форми на car и cdr

Нека $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$.

- `(car l)` $\longrightarrow a_1$
- `(cdr l)` $\longrightarrow (a_2\ a_3 \ldots a_n)$
- `(car (cdr l))` $\longrightarrow a_2 \longleftarrow$ `(cadr l)`
- `(cdr (cdr l))` $\longrightarrow (a_3 \ldots a_n) \longleftarrow$ `(cddr l)`
- `(car (cdr (cdr l)))` $\longrightarrow a_3 \longleftarrow$ `(caddr l)`
- имаме съкратени форми за до 4 последователни прилагания на `car` и `cdr`

Форми на равенство в Scheme

- `(eq?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> <израз$_2$> заемат едно и също място в паметта

Форми на равенство в Scheme

- `(eq?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> <израз$_2$> заемат едно и също място в паметта
- `(eqv?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> заемат едно и също място в паметта или са едни и същи по стойност **атоми** (без функции), дори и да заемат различно място в паметта

Форми на равенство в Scheme

- `(eq?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> <израз$_2$> заемат едно и също място в паметта
- `(eqv?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> заемат едно и също място в паметта или са едни и същи по стойност **атоми** (без функции), дори и да заемат различно място в паметта
  - Ако `(eq?` <израз$_1$> <израз$_2$>`)`,
    то със сигурност `(eqv?` <израз$_1$> <израз$_2$>`)`

Форми на равенство в Scheme

- `(eq?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> <израз$_2$> заемат едно и също място в паметта
- `(eqv?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> заемат едно и също място в паметта или са едни и същи по стойност **атоми** (без функции), дори и да заемат различно място в паметта
  - Ако `(eq?` <израз$_1$> <израз$_2$>`)`,
    то със сигурност `(eqv?` <израз$_1$> <израз$_2$>`)`
- `(equal?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> са едни и същи по стойност **атоми или наредени двойки**, чиито компоненти са равни в смисъла на `equal?`

Форми на равенство в Scheme

- `(eq?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> <израз$_2$> заемат едно и също място в паметта
- `(eqv?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> заемат едно и също място в паметта или са едни и същи по стойност **атоми** (без функции), дори и да заемат различно място в паметта
  - Ако `(eq?` <израз$_1$> <израз$_2$>`)`,
    то със сигурност `(eqv?` <израз$_1$> <израз$_2$>`)`
- `(equal?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> са едни и същи по стойност **атоми или наредени двойки**, чиито компоненти са равни в смисъла на `equal?`
  - В частност, `equal?` проверява за равенство на списъци

Форми на равенство в Scheme

- `(eq?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> <израз$_2$> заемат едно и също място в паметта
- `(eqv?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> заемат едно и също място в паметта или са едни и същи по стойност **атоми** (без функции), дори и да заемат различно място в паметта
  - Ако `(eq?` <израз$_1$> <израз$_2$>`)`,
    то със сигурност `(eqv?` <израз$_1$> <израз$_2$>`)`
- `(equal?` <израз$_1$> <израз$_2$>`)` — връща `#t` точно тогава, когато оценките на <израз$_1$> и <израз$_2$> са едни и същи по стойност **атоми или наредени двойки**, чиито компоненти са равни в смисъла на `equal?`
  - В частност, `equal?` проверява за равенство на списъци
  - Ако `(eqv?` <израз$_1$> <израз$_2$>`)`,
    то със сигурност `(equal?` <израз$_1$> <израз$_2$>`)`

Вградени функции за списъци

- `(length` <списък>`)` — връща дължината на <списък>

Вградени функции за списъци

- `(length` <списък>`)` — връща дължината на <списък>
- `(append` {<списък>}`)` — конкатенира всички <списък>

Вградени функции за списъци

- `(length` <списък>`)` — връща дължината на <списък>
- `(append` {<списък>}`)` — конкатенира всички <списък>
- `(reverse` <списък>`)` — елементите на <списък> в обратен ред

Вградени функции за списъци

- `(length` <списък>`)` — връща дължината на <списък>
- `(append` {<списък>}`)` — конкатенира всички <списък>
- `(reverse` <списък>`)` — елементите на <списък> в обратен ред
- `(list-tail` <списък> `n)` — елементите на <списък> без първите `n`

Вградени функции за списъци

- `(length` <списък>`)` — връща дължината на <списък>
- `(append` {<списък>}`)` — конкатенира всички <списък>
- `(reverse` <списък>`)` — елементите на <списък> в обратен ред
- `(list-tail` <списък> n`)` — елементите на <списък> без първите n
- `(list-ref` <списък> n`)` — n-ти елемент на <списък> (от 0)

Вградени функции за списъци

- (length <списък>) — връща дължината на <списък>
- (append {<списък>}) — конкатенира всички <списък>
- (reverse <списък>) — елементите на <списък> в обратен ред
- (list-tail <списък> n) — елементите на <списък> без първите n
- (list-ref <списък> n) — n-ти елемент на <списък> (от 0)
- (member <елемент> <списък>) — проверява дали <елемент> се среща в <списък>

Вградени функции за списъци

- (length <списък>) — връща дължината на <списък>
- (append {<списък>}) — конкатенира всички <списък>
- (reverse <списък>) — елементите на <списък> в обратен ред
- (list-tail <списък> n) — елементите на <списък> без първите n
- (list-ref <списък> n) — n-ти елемент на <списък> (от 0)
- (member <елемент> <списък>) — проверява дали <елемент> се среща в <списък>
  - По-точно, връща <списък> от първото срещане на <елемент> нататък, ако го има

Вградени функции за списъци

- (length <списък>) — връща дължината на <списък>
- (append {<списък>}) — конкатенира всички <списък>
- (reverse <списък>) — елементите на <списък> в обратен ред
- (list-tail <списък> n) — елементите на <списък> без първите n
- (list-ref <списък> n) — n-ти елемент на <списък> (от 0)
- (member <елемент> <списък>) — проверява дали <елемент> се среща в <списък>
  - По-точно, връща <списък> от първото срещане на <елемент> нататък, ако го има
  - Връща #f, ако <елемент> го няма в <списък>

Вградени функции за списъци

- (length <списък>) — връща дължината на <списък>
- (append {<списък>}) — конкатенира всички <списък>
- (reverse <списък>) — елементите на <списък> в обратен ред
- (list-tail <списък> n) — елементите на <списък> без първите n
- (list-ref <списък> n) — n-ти елемент на <списък> (от 0)
- (member <елемент> <списък>) — проверява дали <елемент> се среща в <списък>
  - По-точно, връща <списък> от първото срещане на <елемент> нататък, ако го има
  - Връща #f, ако <елемент> го няма в <списък>
  - Сравнението на елементи става с equal?

Вградени функции за списъци

- `(length` <списък>`)` — връща дължината на <списък>
- `(append {`<списък>`})` — конкатенира всички <списък>
- `(reverse` <списък>`)` — елементите на <списък> в обратен ред
- `(list-tail` <списък> `n)` — елементите на <списък> без първите `n`
- `(list-ref` <списък> `n)` — `n`-ти елемент на <списък> (от 0)
- `(member` <елемент> <списък>`)` — проверява дали <елемент> се среща в <списък>
  - По-точно, връща <списък> от първото срещане на <елемент> нататък, ако го има
  - Връща `#f`, ако <елемент> го няма в <списък>
  - Сравнението на елементи става с `equal?`
- `(memv` <елемент> <списък>`)` — като `member`, но сравнява с `eqv?`

Вградени функции за списъци

- (length <списък>) — връща дължината на <списък>
- (append {<списък>}) — конкатенира всички <списък>
- (reverse <списък>) — елементите на <списък> в обратен ред
- (list-tail <списък> n) — елементите на <списък> без първите n
- (list-ref <списък> n) — n-ти елемент на <списък> (от 0)
- (member <елемент> <списък>) — проверява дали <елемент> се среща в <списък>
  - По-точно, връща <списък> от първото срещане на <елемент> нататък, ако го има
  - Връща #f, ако <елемент> го няма в <списък>
  - Сравнението на елементи става с equal?
- (memv <елемент> <списък>) — като member, но сравнява с eqv?
- (memq <елемент> <списък>) — като member, но сравнява с eq?

Обхождане на списъци

При обхождане на `l`:

- Ако `l` е празен, връщаме базова стойност **(дъно)**
- Иначе, комбинираме главата `(car l)` с резултата от рекурсивното извикване над опашката `(cdr l)` **(стъпка)**

Обхождане на списъци

При обхождане на `l`:

- Ако `l` е празен, връщаме базова стойност **(дъно)**
- Иначе, комбинираме главата `(car l)` с резултата от рекурсивното извикване над опашката `(cdr l)` **(стъпка)**

Примери: `length`, `list-tail`, `list-ref`, `member`, `memqv`, `memq`

Конструиране на списъци

Използваме рекурсия по даден параметър (напр. число, списък...)

- На дъното връщаме фиксиран списък (например `()`)
- На стъпката построяваме с `cons` списък със съответната глава, а опашката строим чрез рекурсивно извикване на същата функция

Конструиране на списъци

Използваме рекурсия по даден параметър (напр. число, списък...)

- На дъното връщаме фиксиран списък (например `()`)
- На стъпката построяваме с `cons` списък със съответната глава, а опашката строим чрез рекурсивно извикване на същата функция

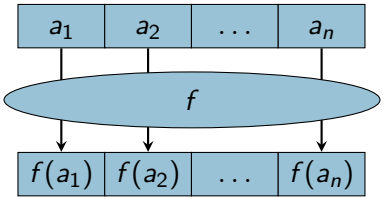
Примери: `from-to`, `collect`, `append`, `reverse`

Изобразяване на списък (map)

Да се дефинира функция (map <функция> <списък>), която връща нов списък съставен от елементите на <списък>, върху всеки от които е приложена <функция>.

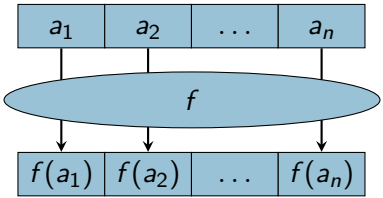
Изобразяване на списък (map)

Да се дефинира функция (map <функция> <списък>), която връща нов списък съставен от елементите на <списък>, върху всеки от които е приложена <функция>.



Изобразяване на списък (map)

Да се дефинира функция (map <функция> <списък>), която връща нов списък съставен от елементите на <списък>, върху всеки от които е приложена <функция>.




```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- (map square '(1 2 3)) $\longrightarrow$?

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` $\longrightarrow$?

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` $\longrightarrow$ `(b e h)`

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` $\longrightarrow$ `(b e h)`
- `(map (lambda (f) (f 2)) (list square 1+ odd?))` $\longrightarrow$?

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` $\longrightarrow$ `(b e h)`
- `(map (lambda (f) (f 2)) (list square 1+ odd?))` $\longrightarrow$ `(4 3 #f)`

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` $\longrightarrow$ `(b e h)`
- `(map (lambda (f) (f 2)) (list square 1+ odd?))` $\longrightarrow$ `(4 3 #f)`
- `(map (lambda (f) (f 2)) (map twice (list square 1+ boolean?)))` $\longrightarrow$?

Изобразяване на списък (map) — примери

```
(define (map f l)
        (if (null? l) '()
            (cons (f (car l)) (map f (cdr l)))))
```

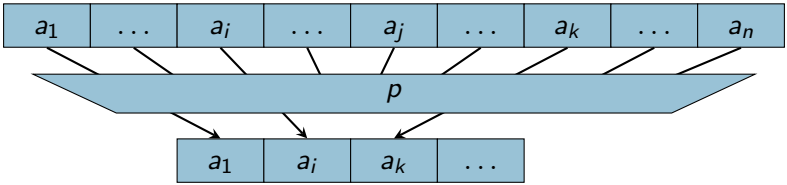
- `(map square '(1 2 3))` $\longrightarrow$ `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` $\longrightarrow$ `(b e h)`
- `(map (lambda (f) (f 2)) (list square 1+ odd?))` $\longrightarrow$ `(4 3 #f)`
- `(map (lambda (f) (f 2)) (map twice (list square 1+ boolean?)))` $\longrightarrow$ `(16 4 #t)`

Филтриране на списък (`filter`)

Да се дефинира функция (`filter` <условие> <списък>), която връща само тези от елементите на <списък>, които удовлетворяват <условие>.

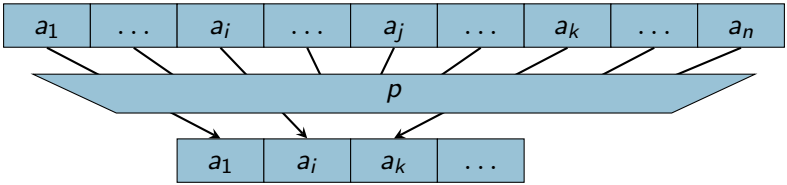
Филтриране на списък (`filter`)

Да се дефинира функция (`filter` <условие> <списък>), която връща само тези от елементите на <списък>, които удовлетворяват <условие>.



Филтриране на списък (filter)

Да се дефинира функция (filter <условие> <списък>), която връща само тези от елементите на <списък>, които удовлетворяват <условие>.




```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$?

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`
- `(filter pair? '((a b) c () d (e)))` $\longrightarrow$?

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`
- `(filter pair? '((a b) c () d (e)))` $\longrightarrow$ `((a b) (e))`

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`
- `(filter pair? '((a b) c () d (e)))` $\longrightarrow$ `((a b) (e))`
- `(map (lambda (x) (filter even? x)) '((1 2 3) (4 5 6) (7 8 9)))`
  $\longrightarrow$?

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`
- `(filter pair? '((a b) c () d (e)))` $\longrightarrow$ `((a b) (e))`
- `(map (lambda (x) (filter even? x)) '((1 2 3) (4 5 6) (7 8 9)))`
  $\longrightarrow$ `((2) (4 6) (8))`

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`
- `(filter pair? '((a b) c () d (e)))` $\longrightarrow$ `((a b) (e))`
- `(map (lambda (x) (filter even? x)) '((1 2 3) (4 5 6) (7 8 9)))` $\longrightarrow$ `((2) (4 6) (8))`
- `(map (lambda (x) (map (lambda (f) (filter f x)) (list negative? zero? positive?))) '((-2 1 0) (1 4 -1) (0 0 1)))` $\longrightarrow$?

Филтриране на списък (filter)

```
(define (filter p? l)
   (cond ((null? l) l)
         ((p? (car l)) (cons (car l) (filter p? (cdr l))))
         (else (filter p? (cdr l)))))
```

- `(filter odd? '(1 2 3 4 5))` $\longrightarrow$ `(1 3 5)`
- `(filter pair? '((a b) c () d (e)))` $\longrightarrow$ `((a b) (e))`
- `(map (lambda (x) (filter even? x)) '((1 2 3) (4 5 6) (7 8 9)))`
  $\longrightarrow$ `((2) (4 6) (8))`
- `(map (lambda (x) (map (lambda (f) (filter f x)) (list negative? zero? positive?))) '((-2 1 0) (1 4 -1) (0 0 1)))`
  $\longrightarrow$ `(((-2) (0) (1)) ((-1) () (1 4)) (() (0 0) (1)))`

Дясно свиване (`foldr`)

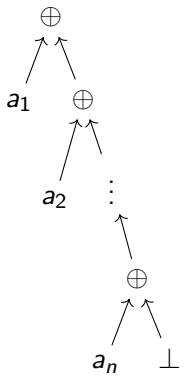
Да се дефинира функция, която по даден списък $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$ пресмята:

$$a_1 \oplus \Big(a_2 \oplus \big(\ldots \oplus (a_n \oplus \bot)\ldots\big)\Big),$$

Дясно свиване (`foldr`)

Да се дефинира функция, която по даден списък $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$ пресмята:

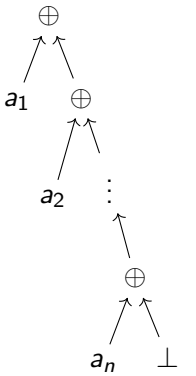
$$a_1 \oplus \Big(a_2 \oplus \big(\ldots \oplus (a_n \oplus \bot)\ldots\big)\Big),$$



Дясно свиване (foldr)

Да се дефинира функция, която по даден списък $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$ пресмята:

$$a_1 \oplus \Big(a_2 \oplus \big(\ldots \oplus (a_n \oplus \bot)\ldots\big)\Big),$$




```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$?

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ `120`

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ `?`

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ `35`

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ 35
- `(foldr cons '() '(1 5 10))` $\longrightarrow$?

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ 35
- `(foldr cons '() '(1 5 10))` $\longrightarrow$ `(1 5 10)`

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ 35
- `(foldr cons '() '(1 5 10))` $\longrightarrow$ `(1 5 10)`
- `(foldr list '() '(1 5 10))` $\longrightarrow$?

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ 35
- `(foldr cons '() '(1 5 10))` $\longrightarrow$ `(1 5 10)`
- `(foldr list '() '(1 5 10))` $\longrightarrow$ `(1 (5 (10 ())))`

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ 35
- `(foldr cons '() '(1 5 10))` $\longrightarrow$ `(1 5 10)`
- `(foldr list '() '(1 5 10))` $\longrightarrow$ `(1 (5 (10 ())))`
- `(foldr append '() '((a b) (c d) (e f)))` $\longrightarrow$?

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` ⟶ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` ⟶ 35
- `(foldr cons '() '(1 5 10))` ⟶ `(1 5 10)`
- `(foldr list '() '(1 5 10))` ⟶ `(1 (5 (10 ())))`
- `(foldr append '() '((a b) (c d) (e f)))` ⟶ `(a b c d e f)`

Дясно свиване (foldr) — примери

```
(define (foldr op nv l)
        (if (null? l) nv
            (op (car l) (foldr op nv (cdr l)))))
```

- `(foldr * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldr + 0 (map square (filter odd? (from-to 1 5))))` $\longrightarrow$ 35
- `(foldr cons '() '(1 5 10))` $\longrightarrow$ `(1 5 10)`
- `(foldr list '() '(1 5 10))` $\longrightarrow$ `(1 (5 (10 ())))`
- `(foldr append '() '((a b) (c d) (e f)))` $\longrightarrow$ `(a b c d e f)`
- `map`, `filter` и `accumulate` могат да се реализират чрез `foldr`

Ляво свиване (`foldl`)

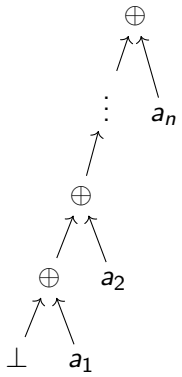
Да се дефинира функция, която по даден списък $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$ пресмята:

$$\Big(\ldots\big((\bot \oplus a_1) \oplus a_2\big) \oplus \ldots\Big) \oplus a_n$$

Ляво свиване (`foldl`)

Да се дефинира функция, която по даден списък $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$ пресмята:

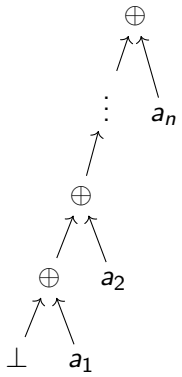
$$\Big(\ldots\big((\bot \oplus a_1) \oplus a_2\big) \oplus \ldots\Big) \oplus a_n$$



Ляво свиване (foldl)

Да се дефинира функция, която по даден списък $l = (a_1\ a_2\ a_3\ \ldots\ a_n)$ пресмята:

$$\Big(\ldots\big((\bot \oplus a_1) \oplus a_2\big) \oplus \ldots\Big) \oplus a_n$$




```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$?

Ляво свиване (`foldl`) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `?`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl ? '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`
- `(foldl list '() '(1 5 10))` $\longrightarrow$ `?`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`
- `(foldl list '() '(1 5 10))` $\longrightarrow$ `((() 1) 5) 10)`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`
- `(foldl list '() '(1 5 10))` $\longrightarrow$ `((() 1) 5) 10)`
- `(foldl append '() '((a b) (c d) (e f)))` $\longrightarrow$?

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ `120`
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`
- `(foldl list '() '(1 5 10))` $\longrightarrow$ `((() 1) 5) 10)`
- `(foldl append '() '((a b) (c d) (e f)))` $\longrightarrow$ `(a b c d e f)`

Ляво свиване (foldl) — примери

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- `(foldl * 1 (from-to 1 5))` $\longrightarrow$ 120
- `(foldl cons '() '(1 5 10))` $\longrightarrow$ `((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` $\longrightarrow$ `(10 5 1)`
- `(foldl list '() '(1 5 10))` $\longrightarrow$ `((() 1) 5) 10)`
- `(foldl append '() '((a b) (c d) (e f)))` $\longrightarrow$ `(a b c d e f)`
- `foldr` генерира линеен рекурсивен процес, а `foldl` — линеен итеративен

Функции от по-висок ред в Racket

В R^5RS е дефинирана само функцията `map`.
В Racket са дефинирани функциите `map`, `filter`, `foldr`, `foldl`

Функции от по-висок ред в Racket

В R^5RS е дефинирана само функцията `map`.
В Racket са дефинирани функциите `map`, `filter`, `foldr`, `foldl`

Внимание: `foldl` в Racket е дефинирана по различен начин!

`foldl` от лекции

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op nv (car l))
             (cdr l))))
```

$$\Big(\ldots\big((\bot \oplus a_1) \oplus a_2\big) \oplus \ldots\Big) \oplus a_n$$

`foldl` в Racket

```
(define (foldl op nv l)
  (if (null? l) nv
      (foldl op (op (car l) nv)
             (cdr l))))
```

$$a_n \oplus \Big(\ldots\big(a_2 \oplus (a_1 \oplus \bot)\big)\ldots\Big),$$

Свиване на непразен списък (`foldr1`, `foldl1`)

Задача. Да се намери максималният елемент на списък.

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max ? l))
```

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max (car l) l))
```

Свиване на непразен списък (`foldr1`, `foldl1`)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max (car l) l))
```

Можем ли да пропуснем нулевата стойност за непразен списък?

Свиване на непразен списък (`foldr1`, `foldl1`)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max (car l) l))
```

Можем ли да пропуснем нулевата стойност за непразен списък?

$$a_1 \oplus \big(\ldots \oplus (a_{n-1} \oplus a_n) \ldots \big)$$

Свиване на непразен списък (`foldr1`, `foldl1`)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max (car l) l))
```

Можем ли да пропуснем нулевата стойност за непразен списък?

$$a_1 \oplus \left(\ldots \oplus \left(a_{n-1} \oplus a_n \right) \ldots \right)$$

```
(define (foldr1 op l)
  (if (null? (cdr l)) (car l)
      (op (car l)
          (foldr1 op (cdr l)))))
```

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max (car l) l))
```

Можем ли да пропуснем нулевата стойност за непразен списък?

$$a_1 \oplus \big(\ldots \oplus (a_{n-1} \oplus a_n) \ldots \big)$$

```
(define (foldr1 op l)
  (if (null? (cdr l)) (car l)
      (op (car l)
          (foldr1 op (cdr l)))))
```

$$\big(\ldots ((a_1 \oplus a_2) \oplus \ldots \big) \oplus a_n$$

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

```
(define (maximum l) (foldr max (car l) l))
```

Можем ли да пропуснем нулевата стойност за непразен списък?

$$a_1 \oplus \left(\ldots \oplus \left(a_{n-1} \oplus a_n \right) \ldots \right)$$

```
(define (foldr1 op l)
  (if (null? (cdr l)) (car l)
      (op (car l)
          (foldr1 op (cdr l)))))
```

$$\left(\ldots \left(\left(a_1 \oplus a_2 \right) \oplus \ldots \right) \oplus a_n \right)$$

```
(define (foldl1 op l)
  (foldl op (car l) (cdr l)))
```

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: ?

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък ()

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък ()
- Вертикално дъно: ?

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък ()
- Вертикално дъно: достигане до друг атом

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък ()
- Вертикално дъно: достигане до друг атом
- Хоризонтална стъпка: ?

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък `()`
- Вертикално дъно: достигане до друг атом
- Хоризонтална стъпка: обхождане на опашката `(cdr l)`

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък `()`
- Вертикално дъно: достигане до друг атом
- Хоризонтална стъпка: обхождане на опашката `(cdr l)`
- Вертикална стъпка: ?

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък `()`
- Вертикално дъно: достигане до друг атом
- Хоризонтална стъпка: обхождане на опашката `(cdr l)`
- Вертикална стъпка: обхождане на главата `(car l)`

Работа с дълбоки списъци

```
((1 (2)) (((3) 4) (5 (6)) () (7)) 8)
```

Задача. Да се преброят в атомите в дълбок списък.
Подход: Обхождане в две посоки: хоризонтално и вертикално

- Хоризонтално дъно: достигане до празен списък `()`
- Вертикално дъно: достигане до друг атом
- Хоризонтална стъпка: обхождане на опашката `(cdr l)`
- Вертикална стъпка: обхождане на главата `(car l)`

За удобство можем да дефинираме функцията `atom?`:

```
(define (atom? x) (and (not (null? x)) (not (pair? x))))
```

Примери

Задача. Да се преброят в атомите в дълбок списък.
`(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8))` $\longrightarrow 8$

Примери

Задача. Да се преброят в атомите в дълбок списък.
`(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8))` $\longrightarrow 8$

```
(define (count-atoms l)
  (cond ((null? l) 0)
        ((atom? l) 1)
        (else (+ (count-atoms (car l)) (count-atoms (cdr l))))))
```

Примери

Задача. Да се преброят в атомите в дълбок списък.
`(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8))` $\longrightarrow$ `8`

```
(define (count-atoms l)
  (cond ((null? l) 0)
        ((atom? l) 1)
        (else (+ (count-atoms (car l)) (count-atoms (cdr l))))))
```

Задача. Да се съберат всички атоми от дълбок списък.
`(flatten '((1 (2)) (((3) 4) (5 (6)) () (7)) 8))` $\longrightarrow$ `(1 2 3 4 5 6 7 8)`

Примери

Задача. Да се преброят в атомите в дълбок списък.
`(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8))` $\longrightarrow$ `8`

```
(define (count-atoms l)
  (cond ((null? l) 0)
        ((atom? l) 1)
        (else (+ (count-atoms (car l)) (count-atoms (cdr l))))))
```

Задача. Да се съберат всички атоми от дълбок списък.
`(flatten '((1 (2)) (((3) 4) (5 (6)) () (7)) 8))` $\longrightarrow$ `(1 2 3 4 5 6 7 8)`

```
(define (flatten l)
  (cond ((null? l) '())
        ((atom? l) (list l))
        (else (append (flatten (car l)) (flatten (cdr l))))))
```

Примери

Задача. Да се обърне редът на атомите в дълбок списък.

```
(deep-reverse '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) ⟶
(8 ((7) () ((6) 5) (4 (3))) ((2) 1))
```

Примери

Задача. Да се обърне редът на атомите в дълбок списък.

```
(deep-reverse '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) ⟶
(8 ((7) () ((6) 5) (4 (3))) ((2) 1))
```

```
(define (deep-reverse l)
  (cond ((null? l) '())
        ((atom? l) l)
        (else (append (deep-reverse (cdr l))
                      (list (deep-reverse (car l)))))))
```

Свиване на дълбоки списъци

```
(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)
```

Свиване на дълбоки списъци

```
(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)
```

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))
```

Свиване на дълбоки списъци

```
(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)
```

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr ? ? ? l))
```

Свиване на дълбоки списъци

```
(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)
```

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + ? ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) ? l))
```

Свиване на дълбоки списъци

```
(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)
```

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr ? ? ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append ? ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append list ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append list '() l))
```

Свиване на дълбоки списъци

```
(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)
```

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append list '() l))



(define (deep-reverse l) (deep-foldr ? ? ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append list '() l))

(define (snoc x l) (append l (list x)))

(define (deep-reverse l) (deep-foldr snoc ? ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append list '() l))

(define (snoc x l) (append l (list x)))

(define (deep-reverse l) (deep-foldr snoc id ? l))
```

Свиване на дълбоки списъци

(deep-foldr <х-дъно> <в-дъно> <операция> <списък>)

```
(define (deep-foldr op term nv l)
  (cond ((null? l) nv)
        ((atom? l) (term l))
        (else (op (deep-foldr op term nv (car l))
                  (deep-foldr op term nv (cdr l))))))

(define (count-atoms l) (deep-foldr + (lambda (x) 1) 0 l))

(define (flatten l) (deep-foldr append list '() l))

(define (snoc x l) (append l (list x)))

(define (deep-reverse l) (deep-foldr snoc id '() l))
```

Директна реализация на `deep-foldr`

Как работи `deep-foldr`?

Директна реализация на `deep-foldr`

Как работи `deep-foldr`?

- пуска себе си рекурсивно за всеки елемент на дълбокия списък

Директна реализация на `deep-foldr`

Как работи `deep-foldr`?

- пуска себе си рекурсивно за всеки елемент на дълбокия списък
- при достигане на вертикално дъно (атоми) прилага `term`

Директна реализация на `deep-foldr`

Как работи `deep-foldr`?

- пуска себе си рекурсивно за всеки елемент на дълбокия списък
- при достигане на вертикално дъно (атоми) прилага `term`
- и събира резултатите с `op`

Директна реализация на `deep-foldr`

Как работи `deep-foldr`?

- пуска себе си рекурсивно за всеки елемент на дълбокия списък
- при достигане на вертикално дъно (атоми) прилага `term`
- и събира резултатите с `op`

Можем да реализираме `deep-foldr` чрез `map` и `foldr`!

Директна реализация на deep-foldr

Как работи deep-foldr?

- пуска себе си рекурсивно за всеки елемент на дълбокия списък
- при достигане на вертикално дъно (атоми) прилага term
- и събира резултатите с op

Можем да реализираме deep-foldr чрез map и foldr!

```
(define (branch p? f g) (lambda (x) (p? x) (f x) (g x)))
(define (deep-foldr op term nv l)
  (foldr op nv
   (map (branch atom?
                term
                (lambda (l) (deep-foldr op term nv l))
        l)))
```

Директна реализация на deep-foldr

Как работи deep-foldr?

- пуска себе си рекурсивно за всеки елемент на дълбокия списък
- при достигане на вертикално дъно (атоми) прилага term
- и събира резултатите с op

Можем да реализираме deep-foldr чрез map и foldr!

```
(define (branch p? f g) (lambda (x) (p? x) (f x) (g x)))
(define (deep-foldr op term nv l)
  (foldr op nv
   (map (branch atom?
                term
                (lambda (l) (deep-foldr op term nv l))
        l)))
```

Задача. Реализирайте функция за ляво свиване на дълбоки списъци deep-foldl.

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри
- `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`)`

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри
- `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`)`
- създава функция с <тяло>, която получава няколко задължителни <параметър> и <списък> от опционални параметри

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри
- `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`)`
- създава функция с <тяло>, която получава няколко задължителни <параметър> и <списък> от опционални параметри
- `(define (`<функция> `.` <списък>`)` <тяло>`)`

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри
- `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`)`
- създава функция с <тяло>, която получава няколко задължителни <параметър> и <списък> от опционални параметри
- `(define (`<функция> `.` <списък>`)` <тяло>`)`
- еквивалентно на
  `(define` <функция> `(lambda` <списък> <тяло>`))`

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри
- `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`)`
- създава функция с <тяло>, която получава няколко задължителни <параметър> и <списък> от опционални параметри
- `(define (`<функция> `.` <списък>`)` <тяло>`)`
- еквивалентно на
  `(define` <функция> `(lambda` <списък> <тяло>`))`
- `(define (`<функция> {<параметър>}$^+$ `.` <списък>`)` <тяло>`)`

Вариадични функции — приемащи произволен брой аргументи

- `(lambda` <списък> <тяло>`)`
- създава функция с <тяло>, която получава <списък> от параметри
- `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`)`
- създава функция с <тяло>, която получава няколко задължителни <параметър> и <списък> от опционални параметри
- `(define (`<функция> `.` <списък>`)` <тяло>`)`
- еквивалентно на
  `(define` <функция> `(lambda` <списък> <тяло>`))`
- `(define (`<функция> {<параметър>}$^+$ `.` <списък>`)` <тяло>`)`
- еквивалентно на
  `(define` <функция> `(lambda (`{<параметър>}$^+$ `.` <списък>`)` <тяло>`))`

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$?

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$?

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$?

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$ Грешка!

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$ Грешка!
- `(define (g x y . l) (append x l y l))`

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$ Грешка!
- `(define (g x y . l) (append x l y l))`
- `(g '(1 2 3) '(4 5 6))` $\longrightarrow$?

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$ Грешка!
- `(define (g x y . l) (append x l y l))`
- `(g '(1 2 3) '(4 5 6))` $\longrightarrow$ `(1 2 3 4 5 6)`

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$ Грешка!
- `(define (g x y . l) (append x l y l))`
- `(g '(1 2 3) '(4 5 6))` $\longrightarrow$ `(1 2 3 4 5 6)`
- `(g '(1 2 3) '(4 5 6) 7 8)` $\longrightarrow$?

Примери

- `(define (maximum x . l) (foldl1 max (cons x l)))`
- `(maximum 7 3 10 2)` $\longrightarrow$ `10`
- `(maximum 100)` $\longrightarrow$ `100`
- `(maximum)` $\longrightarrow$ Грешка!
- `(define (g x y . l) (append x l y l))`
- `(g '(1 2 3) '(4 5 6))` $\longrightarrow$ `(1 2 3 4 5 6)`
- `(g '(1 2 3) '(4 5 6) 7 8)` $\longrightarrow$ `(1 2 3 7 8 4 5 6 7 8)`

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**
- `(apply + '(1 2 3 4 5))` $\longrightarrow$ `15`

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**
- `(apply + '(1 2 3 4 5))` $\longrightarrow$ 15
- `(apply append '((1 2) (3 4) (5 6)))` $\longrightarrow$?

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**
- `(apply + '(1 2 3 4 5))` $\longrightarrow$ `15`
- `(apply append '((1 2) (3 4) (5 6)))` $\longrightarrow$ `(1 2 3 4 5 6)`

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**
- `(apply + '(1 2 3 4 5))` $\longrightarrow$ `15`
- `(apply append '((1 2) (3 4) (5 6)))` $\longrightarrow$ `(1 2 3 4 5 6)`
- `(apply list '(1 2 3 4))` $\longrightarrow$?

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**
- `(apply + '(1 2 3 4 5))` $\longrightarrow$ `15`
- `(apply append '((1 2) (3 4) (5 6)))` $\longrightarrow$ `(1 2 3 4 5 6)`
- `(apply list '(1 2 3 4))` $\longrightarrow$ `(1 2 3 4)`

Прилагане на функция над списък от параметри (apply)

- (apply <функция> <списък>)
- прилага <функция> над <списък> от параметри
- **Примери:**
- (apply + '(1 2 3 4 5)) $\longrightarrow$ 15
- (apply append '((1 2) (3 4) (5 6))) $\longrightarrow$ (1 2 3 4 5 6)
- (apply list '(1 2 3 4)) $\longrightarrow$ (1 2 3 4)

```
(define (append . l)
   (cond ((null? l) '())
         ((null? (car l)) (apply append (cdr l)))
         (else (cons (caar l)
                     (apply append (cons (cdar l) (cdr l)))))))
```

Вариадичен `map`

- Функцията `map` може да се използва с произволен брой списъци!

Вариадичен `map`

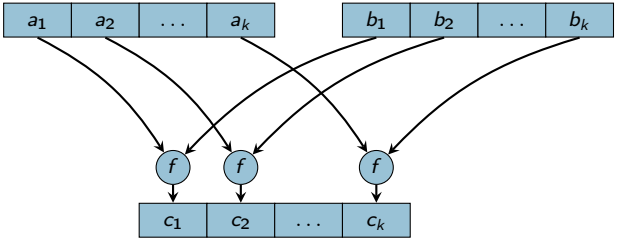
- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`

Вариадичен `map`

- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$

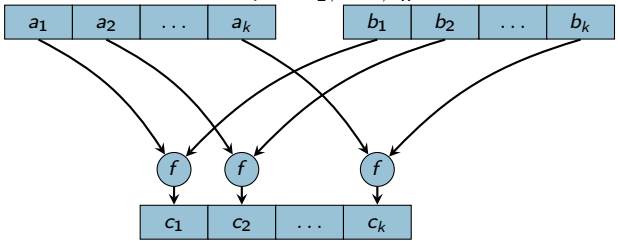
Вариадичен `map`

- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



Вариадичен `map`

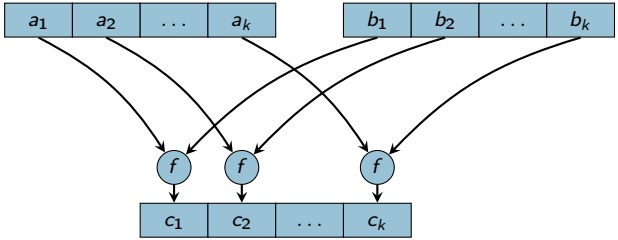
- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



- `(map + '(1 2 3) '(4 5 6))` $\longrightarrow$?

Вариадичен `map`

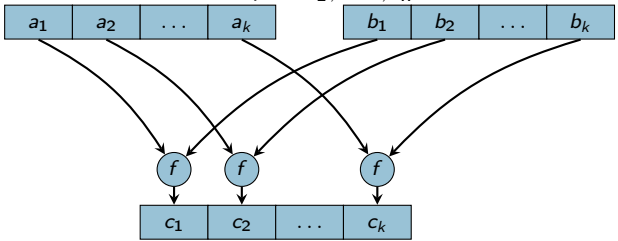
- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



- `(map + '(1 2 3) '(4 5 6))` $\longrightarrow$ `(5 7 9)`

Вариадичен `map`

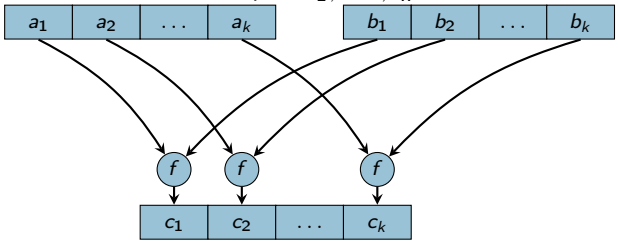
- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



- `(map + '(1 2 3) '(4 5 6))` $\longrightarrow$ `(5 7 9)`
- `(map list '(1 2 3) '(4 5 6))` $\longrightarrow$?

Вариадичен `map`

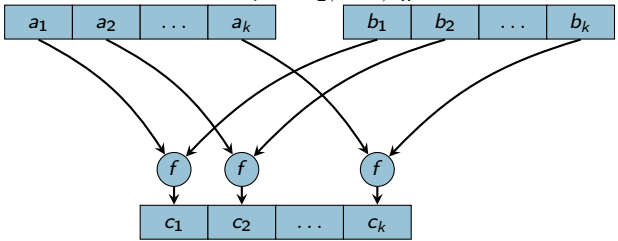
- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



- `(map + '(1 2 3) '(4 5 6))` $\longrightarrow$ `(5 7 9)`
- `(map list '(1 2 3) '(4 5 6))` $\longrightarrow$ `((1 4) (2 5) (3 6))`

Вариадичен `map`

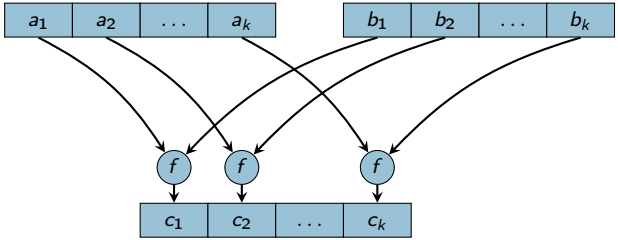
- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



- `(map + '(1 2 3) '(4 5 6))` $\longrightarrow$ `(5 7 9)`
- `(map list '(1 2 3) '(4 5 6))` $\longrightarrow$ `((1 4) (2 5) (3 6))`
- `(map foldr (list * +) '(1 0) '((1 2 3) (4 5 6)))` $\longrightarrow$?

Вариадичен `map`

- Функцията `map` може да се използва с произволен брой списъци!
- `(map` <n-местна функция> $l_1 \ldots l_n$`)`
- Конструира нов списък, като прилага <n-местна функция> над съответните поредни елементи на списъците $l_1, \ldots, l_n$



- `(map + '(1 2 3) '(4 5 6))` $\longrightarrow$ `(5 7 9)`
- `(map list '(1 2 3) '(4 5 6))` $\longrightarrow$ `((1 4) (2 5) (3 6))`
- `(map foldr (list * +) '(1 0) '((1 2 3) (4 5 6)))` $\longrightarrow$ `(6 15)`

Оценяване на списък като комбинация (eval)

- (eval <S-израз> <среда>)

Оценяване на списък като комбинация (eval)

- (eval <S-израз> <среда>)
- връща оценката на <S-израз> в <среда>

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2
- `(evali 'a)` $\longrightarrow$?

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2
- `(evali 'a)` $\longrightarrow$ 2

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2
- `(evali 'a)` $\longrightarrow$ 2
- `(evali ''a)` $\longrightarrow$?

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2
- `(evali 'a)` $\longrightarrow$ 2
- `(evali ''a)` $\longrightarrow$ `a`

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2
- `(evali 'a)` $\longrightarrow$ 2
- `(evali ''a)` $\longrightarrow$ `a`
- `(evali (evali ''a))` $\longrightarrow$?

Оценяване на списък като комбинация (eval)

- `(eval` <S-израз> <среда>`)`
- връща оценката на <S-израз> в <среда>
- `(interaction-environment)` — текущата среда, в която оценяваме
- `(define (evali x) (eval x (interaction-environment)))`
- **Примери:**
- `(define a 2)`
- `a` $\longrightarrow$ 2
- `(evali a)` $\longrightarrow$ 2
- `(evali 'a)` $\longrightarrow$ 2
- `(evali ''a)` $\longrightarrow$ `a`
- `(evali (evali ''a))` $\longrightarrow$ 2

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$?

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ `14`

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ `14`
- `(evali (list 'define b 5))` $\longrightarrow$?

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ `14`
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ `14`
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!
- `(evali (list 'define 'b 5))` $\Longleftrightarrow$ `(define b 5)`

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ `14`
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!
- `(evali (list 'define 'b 5))` $\Longleftrightarrow$ `(define b 5)`
- `b` $\longrightarrow$ `5`

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ 14
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!
- `(evali (list 'define 'b 5))` $\Longleftrightarrow$ `(define b 5)`
- `b` $\longrightarrow$ 5
- `(evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))` $\longrightarrow$?

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ 14
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!
- `(evali (list 'define 'b 5))` $\Longleftrightarrow$ `(define b 5)`
- `b` $\longrightarrow$ 5
- `(evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))` $\longrightarrow$ `a`

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ 14
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!
- `(evali (list 'define 'b 5))` $\Longleftrightarrow$ `(define b 5)`
- `b` $\longrightarrow$ 5
- `(evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))` $\longrightarrow$ a
- `(define (apply f l) (evali (cons f l)))`

Примери за eval

- `(evali (list '+ 5 7 'a))` $\longrightarrow$ `14`
- `(evali (list 'define b 5))` $\longrightarrow$ Грешка!
- `(evali (list 'define 'b 5))` $\Longleftrightarrow$ `(define b 5)`
- `b` $\longrightarrow$ `5`
- `(evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))` $\longrightarrow$ `a`
- `(define (apply f l) (evali (cons f l)))`

Програмите на Scheme могат да се разглеждат като данни!