

Списъци

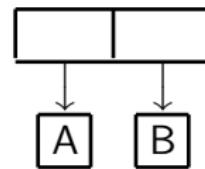
Трифон Трифонов

Функционално програмиране, 2017/18 г.

2–9 ноември 2017 г.

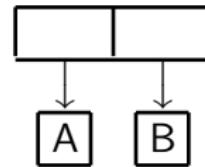
Наредени двойки

(A . B)



Наредени двойки

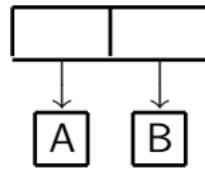
(A . B)



- (cons <израз₁> <израз₂>)

Наредени двойки

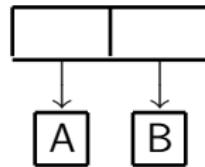
(A . B)



- (`cons <израз1> <израз2>`)
- Наредена двойка от оценките на <израз₁> и <израз₂>

Наредени двойки

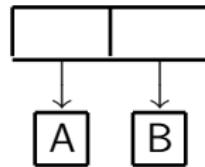
(A . B)



- (`cons <израз1> <израз2>`)
- Наредена двойка от оценките на <израз₁> и <израз₂>
- (`car <израз>`)

Наредени двойки

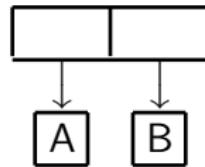
(A . B)



- (`cons <израз1> <израз2>`)
- Наредена двойка от оценките на `<израз1>` и `<израз2>`
- (`car <израз>`)
- **Първият** компонент на двойката, която е оценката на `<израз>`

Наредени двойки

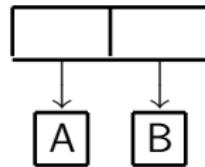
(A . B)



- (`cons <израз1> <израз2>`)
- Наредена двойка от оценките на <израз₁> и <израз₂>
- (`car <израз>`)
- **Първият** компонент на двойката, която е оценката на <израз>
- (`cdr <израз>`)

Наредени двойки

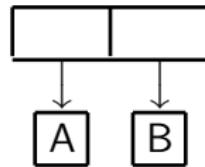
(A . B)



- (`cons <израз1> <израз2>`)
- Наредена двойка от оценките на <израз₁> и <израз₂>
- (`car <израз>`)
- **Първият** компонент на двойката, която е оценката на <израз>
- (`cdr <израз>`)
- **Вторият** компонент на двойката, която е оценката на <израз>

Наредени двойки

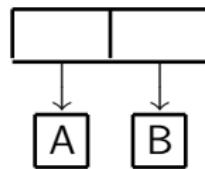
(A . B)



- (`cons <израз1> <израз2>`)
- Наредена двойка от оценките на <израз₁> и <израз₂>
- (`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 <израз1> <израз2>`)
- Наредена двойка от оценките на <израз₁> и <израз₂>
- (`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))

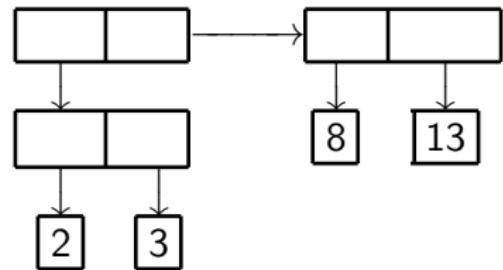


((2 . 3) . (8 . 13))

Примери

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

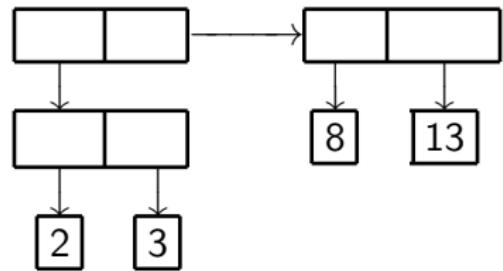
↓
((2 . 3) . (8 . 13))



Примери

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

↓
((2 . 3) . (8 . 13))



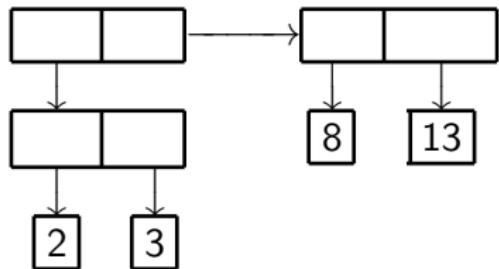
(cons 3 (cons (cons 13 21) 8))

↓
(3 . ((13 . 21) . 8))

Примери

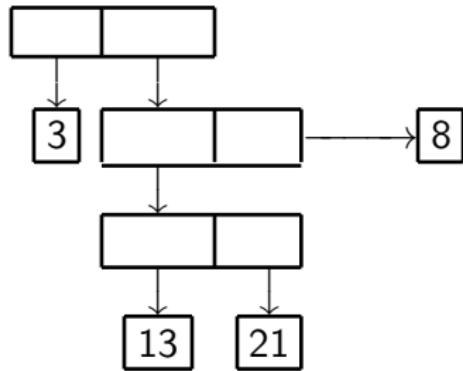
(cons (cons 2 3) (cons 8 13))

↓
((2 . 3) . (8 . 13))



(cons 3 (cons (cons 13 21) 8))

↓
(3 . ((13 . 21) . 8))



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

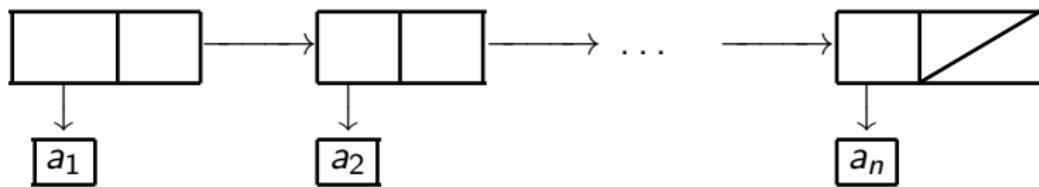
[]

[H | T]

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



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

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 <израз₁> <израз₂> ... <израз_n>)** \iff
`(cons <израз1> (cons <израз2> ... (cons <изразn> '()))))`

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 <израз₁> <израз₂> ... <израз_n>)** \iff
`(cons <израз1> (cons <израз2> ... (cons <изразn> '()))))`
- **(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 <израз1> <израз2> ... <изразn>)`) \iff (`(cons <израз1> (cons <израз2> ... (cons <изразn> '()))))`)
- (`(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 <израз₁> <израз₂> ... <израз_n>) ⇔
(cons <израз₁> (cons <израз₂> ... (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 <израз1> <израз2> ... <изразn>)`) \iff
`(cons <израз1> (cons <израз2> ... (cons <изразn> '()))))`
- (`(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 <израз1> <израз2> ... <изразn>)`) \iff
`(cons <израз1> (cons <израз2> ... (cons <изразn> '()))))`
- (`(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 $l = (a_1 \ a_2 \ a_3 \ \dots \ a_n)$.

- (car l) $\longrightarrow a_1$

Съкратени форми на car и cdr

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

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

Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow ? \longleftarrow (\text{cadr } l)$

Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow a_2 \longleftarrow (\text{cadr } l)$

Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow a_2 \longleftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \longrightarrow ? \longleftarrow (\text{caddr } l)$

Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow a_2 \longleftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \longrightarrow (a_3 \ \dots \ a_n) \longleftarrow (\text{caddr } l)$

Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow a_2 \longleftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \longrightarrow (a_3 \ \dots \ a_n) \longleftarrow (\text{caddr } l)$
- $(\text{car } (\text{cdr } (\text{cdr } l))) \longrightarrow ? \longleftarrow (\text{caddr } l)$

Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow a_2 \longleftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \longrightarrow (a_3 \ \dots \ a_n) \longleftarrow (\text{caddr } l)$
- $(\text{car } (\text{cdr } (\text{cdr } l))) \longrightarrow a_3 \longleftarrow (\text{caddr } l)$

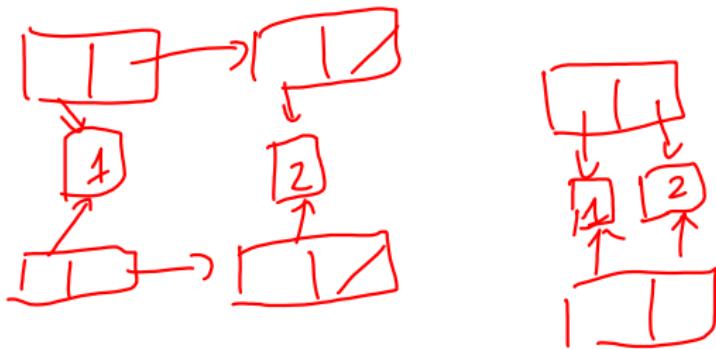
Съкратени форми на car и cdr

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

- $(\text{car } l) \longrightarrow a_1$
- $(\text{cdr } l) \longrightarrow (a_2 \ a_3 \ \dots \ a_n)$
- $(\text{car } (\text{cdr } l)) \longrightarrow a_2 \longleftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \longrightarrow (a_3 \ \dots \ a_n) \longleftarrow (\text{caddr } l)$
- $(\text{car } (\text{cdr } (\text{cdr } l))) \longrightarrow a_3 \longleftarrow (\text{caddr } l)$
- имаме съкратени форми за до 4 последователни прилагания на car и cdr

Форми на равенство в Scheme

- (**eq?** <израз1> <израз2>) — връща #t точно тогава, когато оценките на <израз1> <израз2> заемат едно и също място в паметта



Форми на равенство в Scheme

- (**eq?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> <израз₂> заемат едно и също място в паметта
- (**eqv?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> заемат едно и също място в паметта или са едни и същи по стойност атоми (дори и да заемат различно място в паметта)

Форми на равенство в Scheme

- (**eq?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> <израз₂> заемат едно и също място в паметта
- (**eqv?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> заемат едно и също място в паметта или са едни и същи по стойност атоми (дори и да заемат различно място в паметта)
 - Ако (eq? <израз₁> <израз₂>),
то със сигурност (eqv? <израз₁> <израз₂>)

Форми на равенство в Scheme

- (**eq?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> <израз₂> заемат едно и също място в паметта
- (**eqv?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> заемат едно и също място в паметта или са едни и същи по стойност атоми (дори и да заемат различно място в паметта)
 - Ако (eq? <израз₁> <израз₂>),
то със сигурност (eqv? <израз₁> <израз₂>)
- (**equal?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> са едни и същи по стойност **атоми или наредени двойки**, чийто компоненти са равни в смисъла на equal?

Форми на равенство в Scheme

- (**eq?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> <израз₂> заемат едно и също място в паметта
- (**eqv?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> заемат едно и също място в паметта или са едни и същи по стойност атоми (дори и да заемат различно място в паметта)
 - Ако (eq? <израз₁> <израз₂>),
то със сигурност (eqv? <израз₁> <израз₂>)
- (**equal?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> са едни и същи по стойност **атоми или наредени двойки**, чийто компоненти са равни в смисъла на equal?
 - В частност, equal? проверява за равенство на списъци

Форми на равенство в Scheme

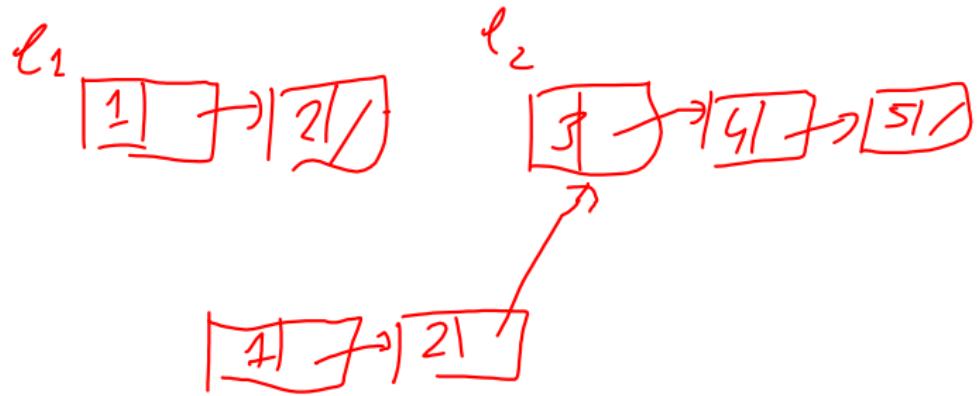
- (**eq?** <израз1> <израз2>) — връща #t точно тогава, когато оценките на <израз1> <израз2> заемат едно и също място в паметта
- (**eqv?** <израз1> <израз2>) — връща #t точно тогава, когато оценките на <израз1> и <израз2> заемат едно и също място в паметта или са едни и същи по стойност атоми (дори и да заемат различно място в паметта)
 - Ако (eq? <израз1> <израз2>),
то със сигурност (eqv? <израз1> <израз2>)
- (**equal?** <израз1> <израз2>) — връща #t точно тогава, когато оценките на <израз1> и <израз2> са едни и същи по стойност **атоми или наредени двойки**, чийто компоненти са равни в смисъла на equal?
 - В частност, equal? проверява за равенство на списъци
 - Ако (eqv? <израз1> <израз2>),
то със сигурност (equal? <израз1> <израз2>)

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

(1 2 3)

(2 3)

(1)

(3)

(2 1)

()

(3 2 1)

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 `<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 `<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 `<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 `<елемент>` нататък, ако го има
 - Връща `#f`, ако `<елемент>` го няма в `<списък>`
 - Сравнението на елементи става с `equal?`
- (`memqv <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 <елемент> нататък, ако го има
 - Връща #f, ако <елемент> го няма в <списък>
 - Сравнението на елементи става с equal?
- (`memqv` <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

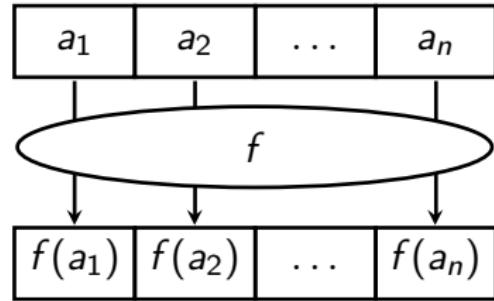
Примери: `from-to`, `collect`, `append`, `reverse`

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>**.

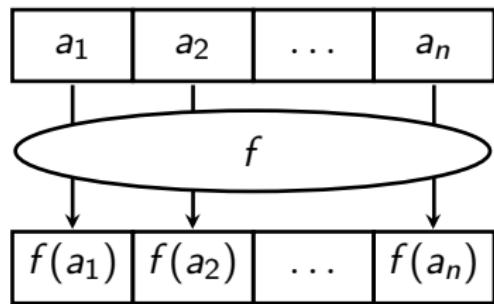
Изобразяване на списък (map)

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)) → ?

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)) → (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)) → (1 4 9)
- (map cadr '((a b c) (d e f) (g h i))) → ?

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

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

($\lambda x \ x^2 + ?$)

(16 4 #t)

Изобразяване на списък (map) — примери

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

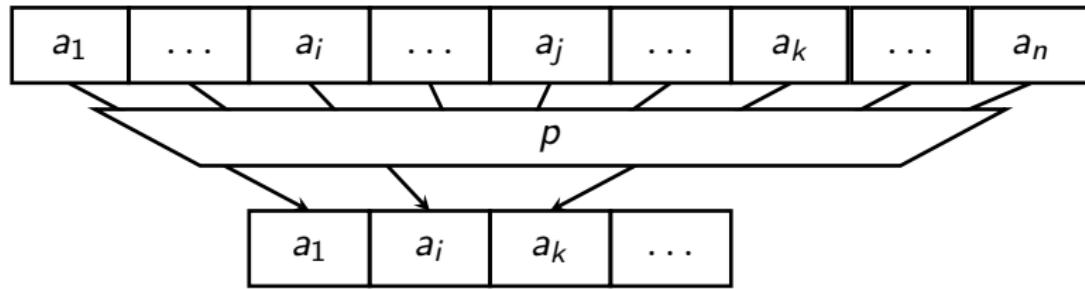
- (`map square '(1 2 3)`) → (1 4 9)
- (`map cadr '((a b c) (d e f) (g h i))`) → (b e h)
- (`map (lambda (f) (f 2)) (list square 1+ odd?)`) → (4 3 #f)
- (`map (lambda (f) (f 2)) (map twice (list square 1+
boolean?)))`) → (16 4 #t)

Филтриране на списък (filter)

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

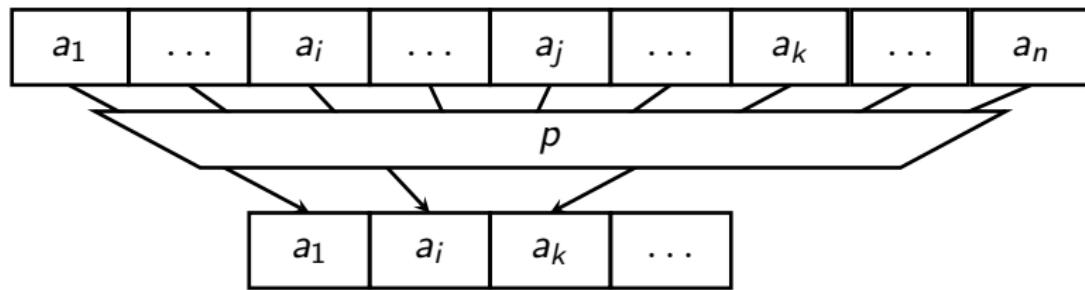
Филтриране на списък (filter)

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)) → ?

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)) → (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)) → (1 3 5)
- (filter pair? '((a b) c () d (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)) → (1 3 5)
- (filter pair? '(({a b) c () d (e})) → ((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)) → (1 3 5)
- (filter pair? '((a b) c () d (e))) → ((a b) (e))
- (map (lambda (x) (filter even? x)) '((1 2 3) (4 5 6) (7 8 9))) → ?

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

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)) → (1 3 5)
- (filter pair? '((a b) c () d (e))) → ((a b) (e))
- (map (lambda (x) (filter even? x)) '((1 2 3) (4 5 6) (7 8 9))) → ((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))) → ((((-2) (0) (1)) ((-1) () (1 4)) ((0) (0 0) (1))))

Дясно свиване (foldr)

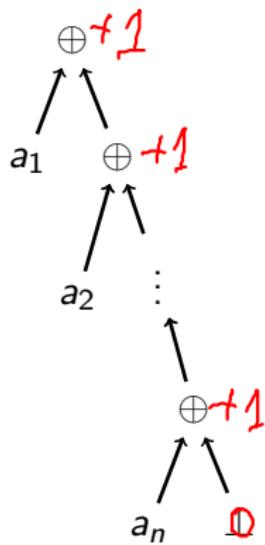
Да се дефинира функция, която по даден списък $l = (a_1 \ a_2 \ a_3 \ \dots \ a_n)$ пресмята:

$$a_1 \oplus \left(a_2 \oplus \left(\dots \oplus (a_n \oplus \perp) \dots \right) \right),$$

Дясно свиване (foldr)

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

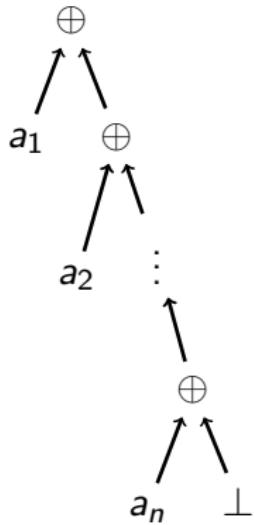
$$a_1 \oplus (a_2 \oplus (\dots \oplus (a_n \oplus \perp) \dots)),$$



Дясно свиване (foldr)

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

$$a_1 \oplus (a_2 \oplus (\dots \oplus (a_n \oplus \perp) \dots)),$$



```
(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)))))
```

- $(\text{foldr} * 1 (\text{from-to } 1 5)) \rightarrow ?$

Дясно свиване (foldr) — примери

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

- $(\text{foldr} * 1 (\text{from-to } 1 5)) \rightarrow 120$

Дясно свиване (foldr) — примери

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

- $(\text{foldr} * 1 (\text{from-to } 1 5)) \rightarrow 120$
- $(\text{foldr} + 0 (\text{map square} (\text{filter odd?} (\text{from-to } 1 5)))) \rightarrow ?$

Дясно свиване (foldr) — примери

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

- $(\text{foldr} * 1 (\text{from-to } 1 5)) \rightarrow 120$
- $(\text{foldr} + 0 (\text{map square} (\text{filter odd?} (\text{from-to } 1 5)))) \rightarrow 35$

Дясно свиване (foldr) — примери

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

- $(\text{foldr} * 1 (\text{from-to } 1 5)) \rightarrow 120$
- $(\text{foldr} + 0 (\text{map square} (\text{filter odd?} (\text{from-to } 1 5)))) \rightarrow 35$
- $(\text{foldr} \text{ cons} '() '(1 5 10)) \rightarrow ?$

Дясно свиване (foldr) — примери

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

- $(\text{foldr} * 1 (\text{from-to } 1 5)) \rightarrow 120$
- $(\text{foldr} + 0 (\text{map square} (\text{filter odd?} (\text{from-to } 1 5)))) \rightarrow 35$
- $(\text{foldr} \text{ cons} '() '(1 5 10)) \rightarrow (1 5 10)$

Дясно свиване (foldr) — примери

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

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

Дясно свиване (foldr) — примери

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

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

Дясно свиване (foldr) — примери

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

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

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)

(append '(& b) (append '(& d) (append '(& 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))` → 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)
- `map`, `filter` и `accumulate` могат да се реализират чрез `foldr`

Ляво свиване (foldl)

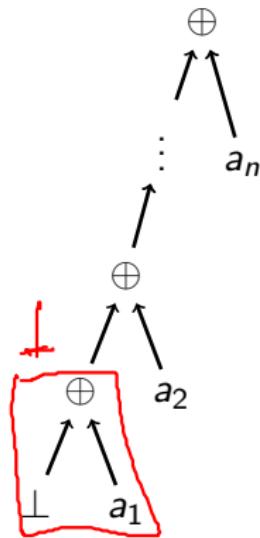
Да се дефинира функция, която по даден списък $l = (a_1 \ a_2 \ a_3 \ \dots \ a_n)$ пресмята:

$$\left(\dots \left((\perp \oplus a_1) \oplus a_2 \right) \oplus \dots \right) \oplus a_n$$

Ляво свиване (foldl)

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

$$(\dots ((\perp \oplus a_1) \oplus a_2) \oplus \dots) \oplus a_n$$



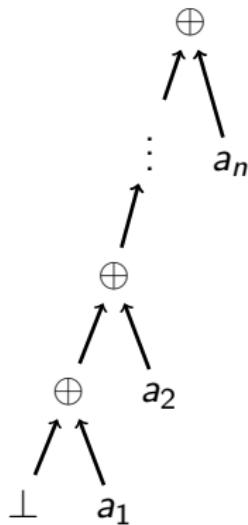
$$(\text{foldr} \cdot - (\text{foldr} \ (\text{foldr} \ '()) a_1) a_2) - \dots - a_n$$

$$(\text{cons} \ a_n - \dots - (\text{cons} \ a_2 \ (\text{cons} \ a_1 \ '()))) - \dots -$$

Ляво свиване (foldl)

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

$$\left(\dots \left((\perp \oplus a_1) \oplus a_2 \right) \oplus \dots \right) \oplus a_n$$



```
(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)) → ?

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)) → 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)) → 120
- (foldl cons '() '(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)) → 120
- (foldl cons '() '(1 5 10)) → ((((). 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)) → 120
- (foldl cons '() '(1 5 10)) → ((((). 1) . 5) . 10)
- (foldl ? '() '(1 5 10)) → (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)) → 120
- (foldl cons '() '(1 5 10)) → ((((). 1) . 5) . 10)
- (foldl (lambda (x y) (cons y x)) '() '(1 5 10)) → (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)) → 120
- (foldl cons '() '(1 5 10)) → ((((). 1) . 5) . 10)
- (foldl (lambda (x y) (cons y x)) '() '(1 5 10)) → (10 5 1)
- (foldl list '() '(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)) → 120
- (foldl cons '() '(1 5 10)) → (((() . 1) . 5) . 10)
- (foldl (lambda (x y) (cons y x)) '() '(1 5 10)) → (10 5 1)
- (foldl list '() '(1 5 10)) → (((() 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)) → 120
- (foldl cons '() '(1 5 10)) → (((() . 1) . 5) . 10)
- (foldl (lambda (x y) (cons y x)) '() '(1 5 10)) → (10 5 1)
- (foldl list '() '(1 5 10)) → (((() 1) 5) 10)
- (foldl append '() '((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)) → 120
- (foldl cons '() '(1 5 10)) → (((() . 1) . 5) . 10)
- (foldl (lambda (x y) (cons y x)) '() '(1 5 10)) → (10 5 1)
- (foldl list '() '(1 5 10)) → (((() 1) 5) 10)
- (foldl append '() '((a b) (c d) (e f))) → (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)) → 120
- (foldl cons '() '(1 5 10)) → (((() . 1) . 5) . 10)
- (foldl (lambda (x y) (cons y x)) '() '(1 5 10)) → (10 5 1)
- (foldl list '() '(1 5 10)) → (((() 1) 5) 10)
- (foldl append '() '((a b) (c d) (e f))) → (a b c d e f)
- foldr генерира линеен рекурсивен процес, а foldl — линеен итеративен

Функции от по-висок ред в Racket

В R⁵RS е дефинирана само функцията `map`.

В Racket са дефинирани функциите `map`, `filter`, `foldr`, `foldl`

Функции от по-висок ред в Racket

В R⁵RS 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))))
```

$$\left(\dots ((\perp \oplus a_1) \oplus a_2) \oplus \dots \right) \oplus a_n$$

`foldl` в Racket

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

$$a_n \oplus \left(\dots (a_2 \oplus (a_1 \oplus \perp)) \dots \right),$$

Сиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

Сиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

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

Сиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

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

Сиване 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?

Сиване 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?

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

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?

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

```
(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?

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

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

$$(\dots ((a_1 \oplus a_2) \oplus \dots) \oplus a_n$$

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?

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

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

$$(\dots ((a_1 \oplus a_2) \oplus \dots) \oplus a_n$$

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

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен списък.

Подход: Обхождане в две посоки: хоризонтално и вертикално

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен списък.

Подход: Обхождане в две посоки: хоризонтално и вертикално

- **Хоризонтално дъно:** ?

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен списък.

Подход: Обхождане в две посоки: хоризонтално и вертикално

- **Хоризонтално дъно:** достигане до празен списък ()

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен списък.

Подход: Обхождане в две посоки: хоризонтално и вертикално

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

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен списък.

Подход: Обхождане в две посоки: хоризонтално и вертикално

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

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен списък.

Подход: Обхождане в две посоки: хоризонтално и вертикално

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

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен 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 1)

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен 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 1)
- **Вертикална стъпка:** ?

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен 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 1)
- **Вертикална стъпка:** обхождане на главата (car 1)

Работа с вложени списъци

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

Задача. Да се преброят в атомите във вложен 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 1)
- **Вертикална стъпка:** обхождане на главата (car 1)

За удобство можем да дефинираме функцията atom?:

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

Примери

Задача. Да се преброят в атомите във вложен списък.

(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) → 8

Примери

Задача. Да се преброят в атомите във вложен списък.

(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) → 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 във вложен списък.

```
(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) —→ 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 вложен списък.

```
(flatten '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) —→
(1 2 3 4 5 6 7 8)
```

Примери

Задача. Да се преброят в атомите във вложен списък.

```
(count-atoms '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) —→ 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 вложен списък.

```
(flatten '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) —→
(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 във вложен 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 във вложен 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 вложени списъци

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

Свиване на вложени списъци

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

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

Свиване на вложени списъци

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

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

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

Свиване на вложени списъци

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

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

(define (count-atoms l) (deep-fold 0 ? ? l))
```

Свиване на вложени списъци

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

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

(define (count-atoms l) (deep-fold 0 (lambda (x) 1) ? 1))
```

Свиване на вложени списъци

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

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

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

Сиване на вложени списъци

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

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

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

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

Свиване на вложени списъци

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

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

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

(define (flatten l) (deep-fold '() ? ? l))
```

Сиване на вложени списъци

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

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

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

(define (flatten l) (deep-fold '() list ? l))
```

Сиване на вложени списъци

```
(deep-fold <x-дъно> <в-дъно> <операция> <списък>)

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

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

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

Свиване на вложени списъци

```
(deep-fold <x-дъно> <в-дъно> <операция> <списък>)

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

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

(define (flatten l) (deep-fold '() list append 1))

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

Свиване на вложени списъци

```
(deep-fold <x-дъно> <в-дъно> <операция> <списък>)

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

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

(define (flatten l) (deep-fold '() list append 1))

(define (deep-reverse l) (deep-fold '() ? ? l))
```

Свиване на вложени списъци

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

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

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

(define (flatten l) (deep-fold '() list append 1))

(define (deep-reverse l) (deep-fold '() id ? l))
```

Свиване на вложени списъци

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

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

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

(define (flatten l) (deep-fold '() list append 1))

(define (rcons x l) (append l (list x)))

(define (deep-reverse l) (deep-fold '() id rcons l))
```

Директна реализация на deep-fold

Как работи deep-fold?

Директна реализация на deep-fold

Как работи deep-fold?

- пуска себе си рекурсивно за всеки елемент на вложенния списък

Директна реализация на deep-fold

Как работи deep-fold?

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

Директна реализация на deep-fold

Как работи deep-fold?

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

Директна реализация на deep-fold

Как работи deep-fold?

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

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

Директна реализация на deep-fold

Как работи deep-fold?

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

Можем да реализираме deep-fold чрез map и foldr!

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

Приемане на произволен брой аргументи

- (lambda <списък> <тяло>)

Приемане на произволен брой аргументи

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

Приемане на произволен брой аргументи

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

Приемане на 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 optionalни параметри

Приемане на 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 optionalни параметри
- `(define (<функция> . <списък>) <тяло>)`

Приемане на 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 optionalни параметри
- `(define (<функция> . <списък>) <тяло>)`
- еквивалентно на
`(define <функция> (lambda <списък> <тяло>))`

Приемане на 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 optionalни параметри
- **(define (<функция> . <списък>) <тяло>)**
- еквивалентно на
`(define <функция> (lambda <списък> <тяло>))`
- **(define (<функция> {<параметър>}⁺ . <списък>) <тяло>)**

Приемане на 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 optionalни 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 . 1) (foldl1 max (cons x 1)))`)
- (`(maximum 7 3 10 2)` → ?

Примери

- `(define (maximum x . 1) (foldl1 max (cons x 1)))`
- `(maximum 7 3 10 2) → 10`

Примери

- (`define (maximum x . 1) (foldl1 max (cons x 1)))`)
- (`(maximum 7 3 10 2)`) → 10
- (`(maximum 100)`) → ?

Примери

- (`define (maximum x . 1) (foldl1 max (cons x 1)))`)
- (`(maximum 7 3 10 2)`) → 10
- (`(maximum 100)`) → 100

Примери

- (`define (maximum x . 1) (foldl1 max (cons x 1)))`)
- (`(maximum 7 3 10 2)`) → 10
- (`(maximum 100)`) → 100
- (`(maximum)`) → ?

Примери

- (`define (maximum x . 1) (foldl1 max (cons x 1)))`)
- (`(maximum 7 3 10 2)`) → 10
- (`(maximum 100)`) → 100
- (`(maximum)`) → Грешка!

Примери

- (`define` (`maximum` `x` . `1`) (`foldl1` `max` (`cons` `x` `1`)))
- (`maximum` `7` `3` `10` `2`) → `10`
- (`maximum` `100`) → `100`
- (`maximum`) → Грешка!
- (`define` (`g` `x` `y` . `1`) (`append` (`append` `x` `1`) (`append` `y` `1`)))

Примери

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

Примери

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

Примери

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

Примери

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

map с произволен брой аргументи

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

map с произволен брой аргументи

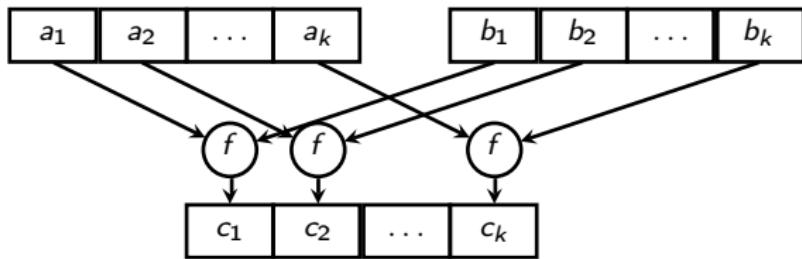
- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)

map с произволен брой аргументи

- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n

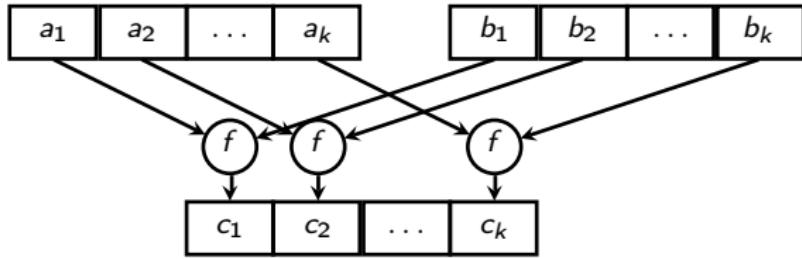
map с произволен брой аргументи

- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



map с произволен брой аргументи

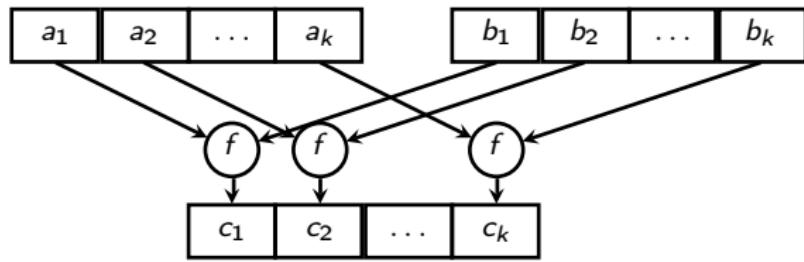
- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



- `(map + '(1 2 3) '(4 5 6))` → ?

map с произволен брой аргументи

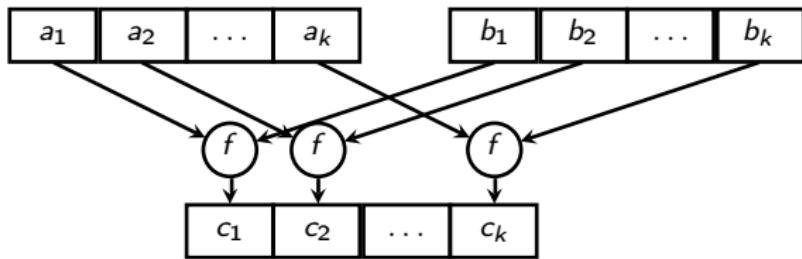
- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



- `(map + '(1 2 3) '(4 5 6)) → (5 7 9)`

map с произволен брой аргументи

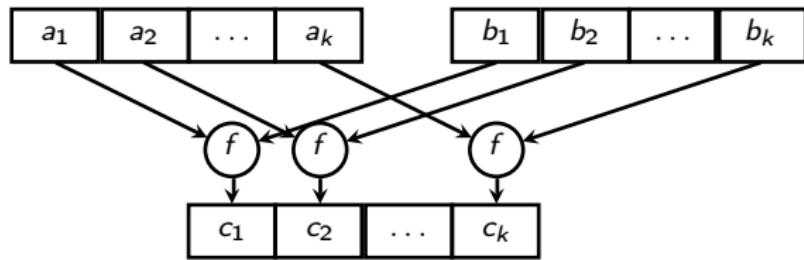
- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



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

map с произволен брой аргументи

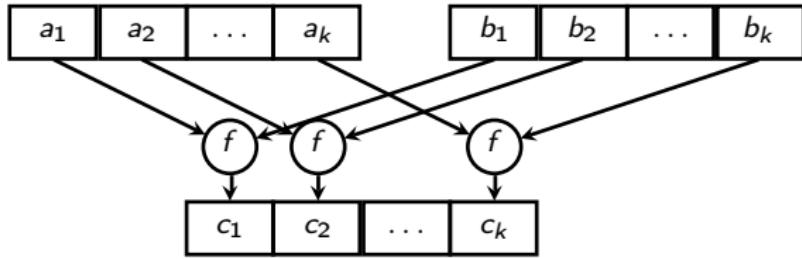
- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



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

map с произволен брой аргументи

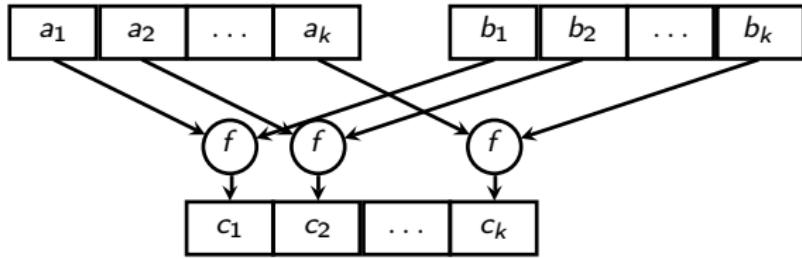
- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



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

map с произволен брой аргументи

- Функцията `map` може да се използва с произволен брой списъци!
- (`map <n-местна функция> l1 ... ln`)
- Конструира нов списък, като прилага `<n-местна функция>` над съответните поредни елементи на списъците l_1, \dots, l_n



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

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))` → 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)) → 15
- (apply append '((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))` → 15
- `(apply append '((1 2) (3 4) (5 6)))` → (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))` → 15
- `(apply append '((1 2) (3 4) (5 6)))` → (1 2 3 4 5 6)
- `(apply list '(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))` → 15
- `(apply append '((1 2) (3 4) (5 6)))` → (1 2 3 4 5 6)
- `(apply list '(1 2 3 4))` → (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)) → 15
- (apply append '((1 2) (3 4) (5 6))) → (1 2 3 4 5 6)
- (apply list '(1 2 3 4)) → (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)))))))
```

Оценяване на списък като комбинация (eval)

- **(eval <списък> <среда>)**

Оценяване на списък като комбинация (eval)

- **(eval <списък> <среда>)**
- връща оценката на **оценката на <списък>** в <среда>

Оценяване на списък като комбинация (eval)

- **(eval <списък> <среда>)**
- връща оценката на **оценката на <списък>** в **<среда>**
- **(interaction-environment)** — текущата среда, в която оценяваме

Оценяване на списък като комбинация (`eval`)

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

Оценяване на списък като комбинация (eval)

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

Оценяване на списък като комбинация (eval)

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

Оценяване на списък като комбинация (eval)

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

Оценяване на списък като комбинация (eval)

- **(eval <списък> <среда>)**
- връща оценката на **оценката на <списък>** в **<среда>**
- **(interaction-environment)** — текущата среда, в която оценяваме
- **(define (evali x) (eval x (interaction-environment)))**
- Примери:
- **(define a 2)**
- **a → 2**
- **(evali a) → 2**
- **(evali 'a) → ?**

Оценяване на списък като комбинация (eval)

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

Оценяване на списък като комбинация (eval)

- **(eval <списък> <среда>)**
- връща оценката на **оценката на** **<списък>** в **<среда>**
- **(interaction-environment)** — текущата среда, в която оценяваме
- **(define (evali x) (eval x (interaction-environment)))**
- Примери:
- **(define a 2)**
- **a → 2**
- **(evali a) → 2**
- **(evali 'a) → 2**
- **(evali ~~'~~a) → ?**

Оценяване на списък като комбинация (eval)

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

Оценяване на списък като комбинация (eval)

- **(eval <списък> <среда>)**
- връща оценката на **оценката на** **<списък>** в **<среда>**
- **(interaction-environment)** — текущата среда, в която оценяваме
- **(define (evali x) (eval x (interaction-environment)))**
- Примери:
- **(define a 2)**
- **a → 2**
- **(evali a) → 2**
- **(evali 'a) → 2**
- **(evali ''a) → a**
- **(evali (evali // a)) → ?**

Оценяване на списък като комбинация (eval)

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

Примери за eval

- (evali (list '+ 5 7 a)) → ?

Примери за eval

- (evali (list '+ 5 7 a)) → 14

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → ?

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!
- (evali (list 'define 'b 5)) ⇔ (define b 5)

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!
- (evali (list 'define 'b 5)) ⇔ (define b 5)
- b → 5

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!
- (evali (list 'define 'b 5)) ⇔ (define b 5)
- b → 5
- (evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))) → ?

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!
- (evali (list 'define 'b 5)) ⇔ (define b 5)
- b → 5
- (evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))) → a

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!
- (evali (list 'define 'b 5)) ⇔ (define b 5)
- b → 5
- (evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))) → a
- (define (apply f l) (evali (cons f l)))

Примери за eval

- (evali (list '+ 5 7 a)) → 14
- (evali (list 'define b 5)) → Грешка!
- (evali (list 'define 'b 5)) ⇔ (define b 5)
- b → 5
- (evali (list 'if (list '< 2 5) (list 'quote 'a) 'b))) → a
- (define (apply f l) (evali (cons f l)))

Програмите на Scheme могат да се разглеждат като данни!