

Списъци

Трифон Трифонов

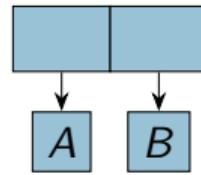
Функционално програмиране, 2024/25 г.

16–23 октомври 2024 г.

Тази презентация е достъпна под лиценза Creative Commons Признание-Некомерсиално-Споделяне на споделеното 4.0 Международен 

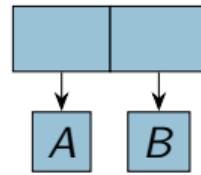
Наредени двойки

(A . B)



Наредени двойки

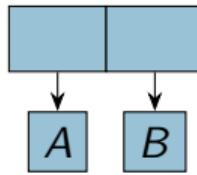
(A . B)



- (**cons** <израз1> <израз2>)

Наредени двойки

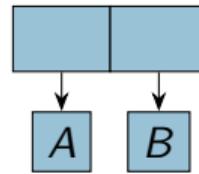
(A . B)



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

Наредени двойки

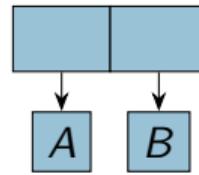
(A . B)



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

Наредени двойки

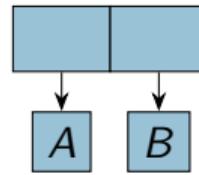
(A . B)



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

Наредени двойки

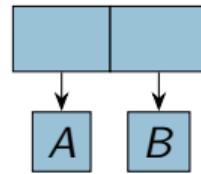
(A . B)



- (**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

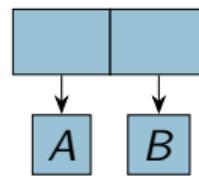
(A . B)



- (**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

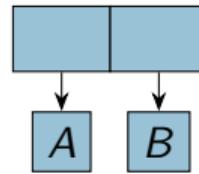
(A . B)



- (**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))
```



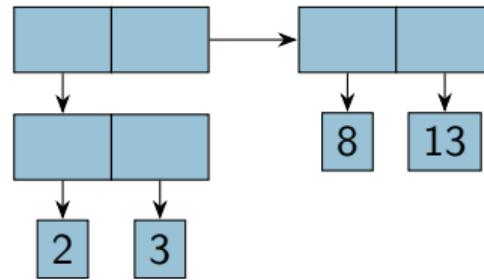
```
((2 . 3) . (8 . 13))
```

Примери

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



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

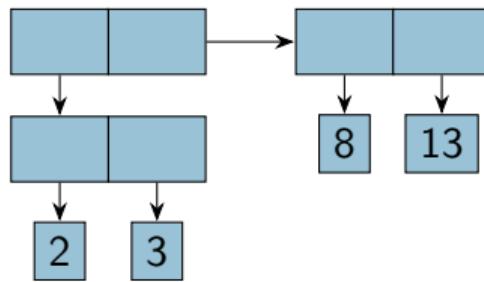


Примери

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



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



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



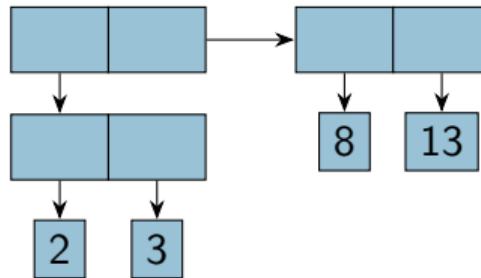
(3 . ((13 . 21) . 8))

Примери

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



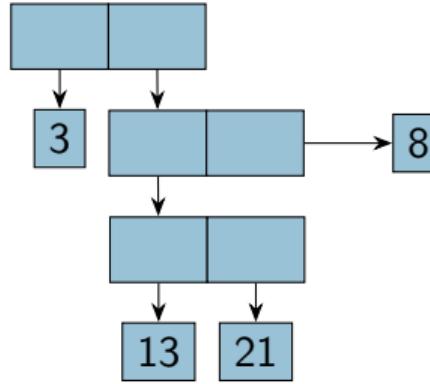
((2 . 3) . (8 . 13))



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



(3 . ((13 . 21) . 8))



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.

Списъци в Scheme

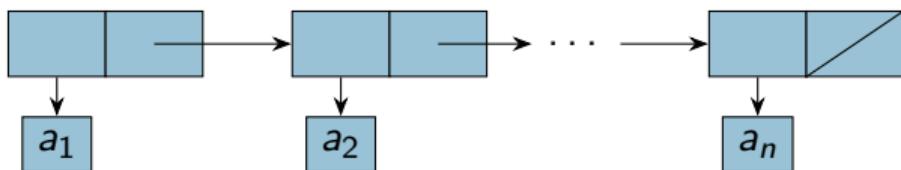
Дефиниция

- 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



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

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

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>
- **(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> ... <израз_n>)** \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

[H | T]

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

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

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

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

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>` и `<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** (без функции), 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** (без функции), 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** (без функции), 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**, чиито 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** (без функции), 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**, чиито компоненти са равни в смисъла на equal?
 - В частност, equal? проверява за равенство на списъци

Форми на равенство в Scheme

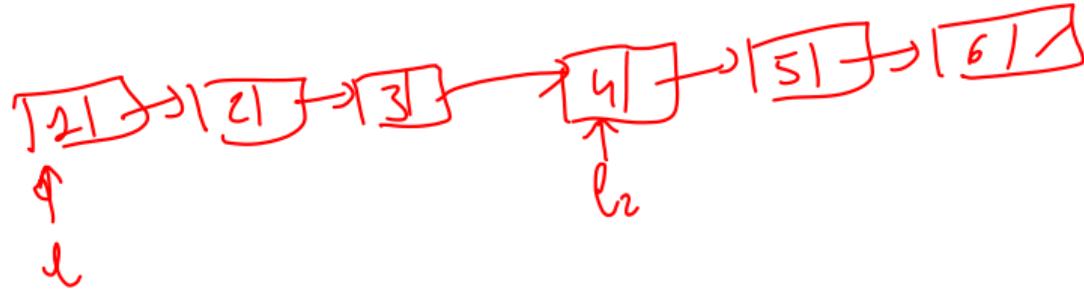
- (**eq?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> <израз₂> заемат едно и също място в паметта
- (**eqv?** <израз₁> <израз₂>) — връща #t точно тогава, когато оценките на <израз₁> и <израз₂> заемат едно и също място в паметта или са едни и същи по стойност **атоми** (без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

$(1 \underline{2} \ 3)$ $(3 \ 2) \quad (1)$
 $(cons \ <\text{ел.}> \ <\text{сн.}>)$?
 $(list \ <\text{ел.}> \ <\text{ел.}>)$
 $(current \ <\text{ел.}> \ <\text{сн.}>)$

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)
 - 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?`

$\text{#t} \leftarrow \text{f} \& x = x$
 $\text{#f} \leftarrow \text{f} \& x = \text{#f}$

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

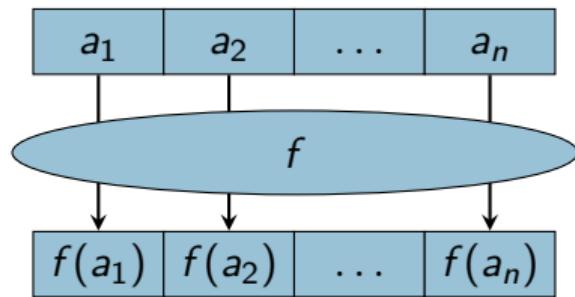
Примери: `from-to`, `collect`, `append`, `reverse`

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>**.

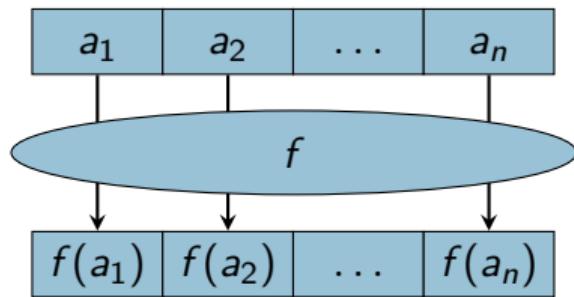
Изобразяване на списък (map)

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)))))
```

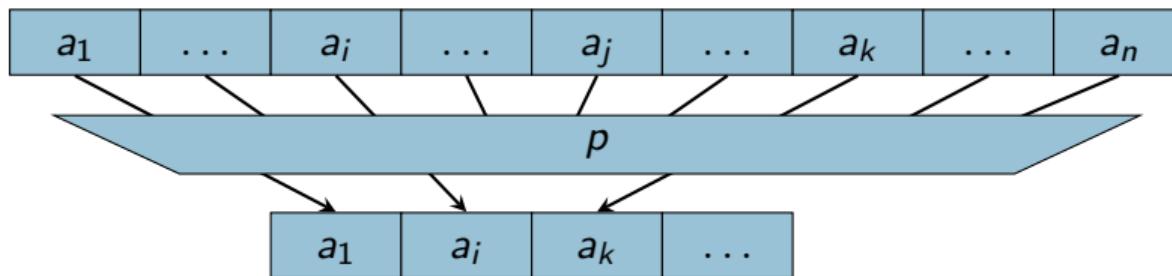
- (`map square '(1 2 3)`) → (1 4 9)
- (`map cadr '((a b c) (d e f) (g h i))`) → (b e h)
- (`map (lambda (f) (f 2)) (list square 1+ odd?)`) → (4 3 #f)

Филтриране на списък (filter)

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

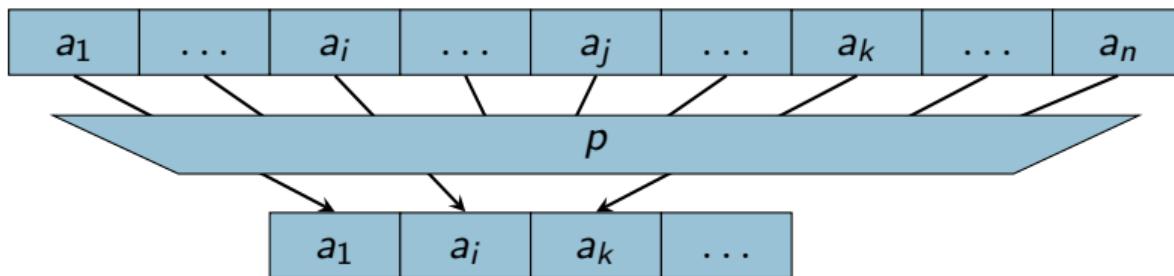
Филтриране на списък (filter)

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) (1))))

Дясно свиване (foldr)

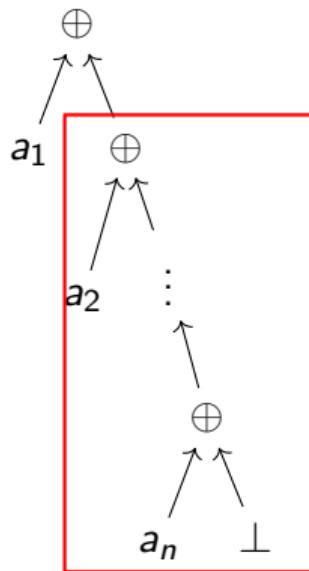
Да се дефинира функция, която по даден списък $l = (a_1 \ a_2 \ a_3 \ \dots \ a_n)$ пресмята:

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

Дясно свиване (foldr)

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

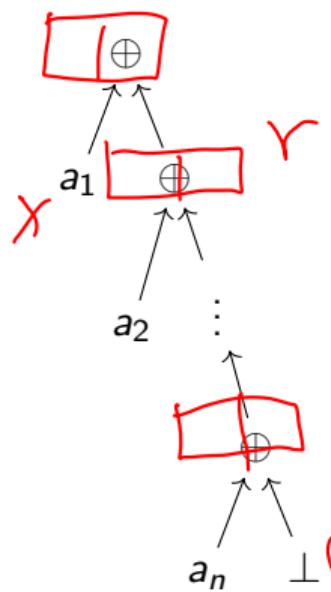
$$a_1 \oplus \left(a_2 \oplus \left(\dots \oplus (a_n \oplus \perp) \dots \right) \right),$$



Дясно свиване (foldr)

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

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



```
(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

```
(1 . (5 , (10 . ())))  

(define (foldr op nv 1)
  (if (null? l) nv
      (op (car l) (foldr op nv (cdr l)))))  

  (list 1 (list 5 (list 10 '()))))
```

- (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 ())))

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)))))
```

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

Дясно свиване (foldr) — примери

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

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

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

Ляво свиване (foldl)

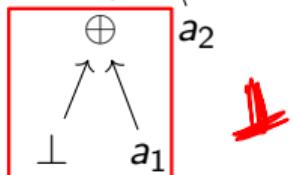
Да се дефинира функция, която по даден списък $l = (a_1 \ a_2 \ a_3 \ \dots \ a_n)$ пресмята:

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



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

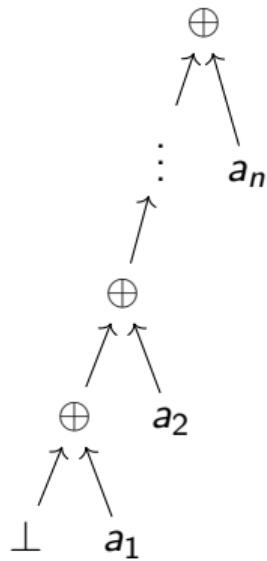
$$(a + b) + c = a + (b + c)$$



Ляво свиване (foldl)

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

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



```
(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 1)
  (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 1)
  (if (null? l) nv
      (foldl op (op nv (car l)) (cdr l))))
```

- (foldl * 1 (from-to 1 5)) → 120
- (foldl cons '() '(1 5 10)) → ?

(cons (cons (cons '() 1) 5) 10)

Ляво свиване (foldl) — примери

```
(define (foldl op nv 1)
  (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 1)
  (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)

() (1) (2 1) (3 2 1)

Ляво свиване (foldl) — примери

```
(define (foldl op nv 1)
  (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 1)
  (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 1)
  (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 1)
  (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 1)
  (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?

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

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?

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

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

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

```
(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))))
```

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:
 - (+ 1 2 3 4 5)
 - (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) (list 1 2 3 4)

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 → 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` → 2
 - (`(evali 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` → 2
 - (`evali a`) → 2
 - (`evali /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` → 2
 - (`evali a`) → 2
 - (`evali '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` → 2
 - (`evali a`) → 2
 - (`evali 'a`) → 2
 - (~~`evali / '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` → 2
 - (`evali a`) → 2
 - (`evali 'a`) → 2
 - (`evali ''a`) → 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 → 2**
 - **(evali a) → 2**
 - **(evali 'a) → 2**
 - **(evali ''a) → a**
 - **(evali (evali //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 → 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))) → ?

(if (< 2 5) (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!