

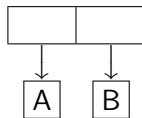
Списъци

Трифон Трифонов

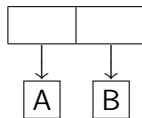
Функционално програмиране, спец. Информатика, 2016/17 г.

3 ноември 2016 г.

Точкови двойки

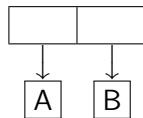
 $(A \cdot B)$ 

Точкови двойки

 $(A . B)$ 

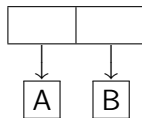
- `(cons <израз1> <израз2>)`

Точкови двойки

 $(A . B)$ 

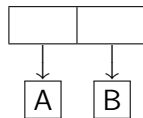
- `(cons <израз1> <израз2>)`
- Точкова двойка от оценките на `<израз1>` и `<израз2>`

Точкови двойки

 $(A . B)$ 

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

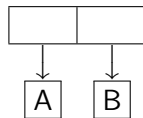
Точкови двойки

 $(A . B)$ 

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

Точкови двойки

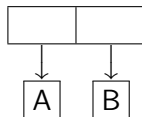
(A . B)



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

Точкови двойки

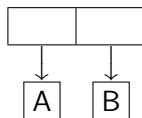
(A . B)



- (cons <израз₁> <израз₂>)
- Точкова 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>

Точкови двойки

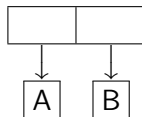
(A . B)



- (cons <израз₁> <израз₂>)
- Точкова 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>)

Точкови двойки

(A . B)



- (cons <израз₁> <израз₂>)
- Точкова 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 точкова двойка

Примери

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



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

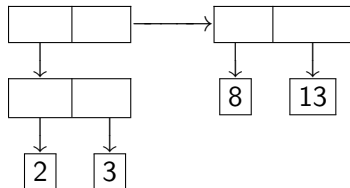
Примери

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

```

      ↓
((2 . 3) . (8 . 13))

```



Примери

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

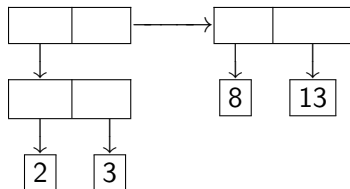
↓

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

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

↓

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



Примери

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

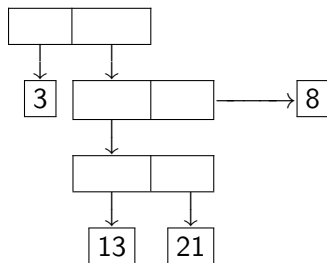
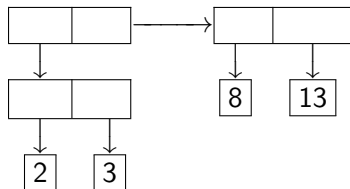
↓

```
((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)
- точкови 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)
- точкови двойки ($S_1 . S_2$), където S_1 и S_2 са S-изрази

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

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

All you need is λ — точкови двойки

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

All you need is λ — точкови 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 — точкови двойки

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

Вариант №1:

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

Вариант №2:

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

Списъци в Scheme

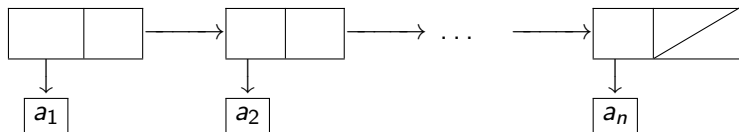
Дефиниция

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

Списъци в Scheme

Дефиниция

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



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

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> ... <изразn>)` \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> ... <изразn>)` \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> ... <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>

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 точкова 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 точкова двойка!
- `(car '())` \rightarrow Грешка!, `(cdr '())` \rightarrow Грешка!

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

- $(\text{car } l) \rightarrow a_1$

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

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

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

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

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

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

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

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

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

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

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

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

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

- $(\text{car } l) \rightarrow a_1$
- $(\text{cdr } l) \rightarrow (a_2 a_3 \dots a_n)$
- $(\text{car } (\text{cdr } l)) \rightarrow a_2 \leftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \rightarrow (a_3 \dots a_n) \leftarrow (\text{cddr } l)$
- $(\text{car } (\text{cdr } (\text{cdr } l))) \rightarrow a_3 \leftarrow (\text{caddr } l)$

Съкратени форми на car и cdr

Нека $l = (a_1 a_2 a_3 \dots a_n)$.

- $(\text{car } l) \rightarrow a_1$
- $(\text{cdr } l) \rightarrow (a_2 a_3 \dots a_n)$
- $(\text{car } (\text{cdr } l)) \rightarrow a_2 \leftarrow (\text{cadr } l)$
- $(\text{cdr } (\text{cdr } l)) \rightarrow (a_3 \dots a_n) \leftarrow (\text{cddr } l)$
- $(\text{car } (\text{cdr } (\text{cdr } l))) \rightarrow a_3 \leftarrow (\text{caddr } l)$
- 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 **точкови 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** (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 **точкови 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** (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 **точкови 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

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?`
- `(memq <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

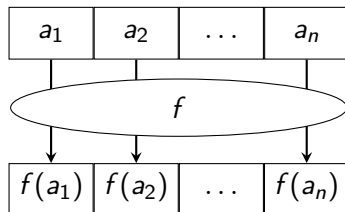
Примери: `from-to`, `collect`, `append`, `reverse`

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>`.

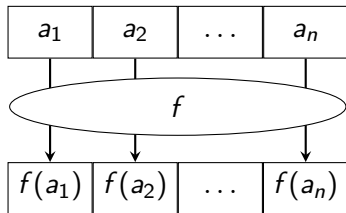
Изобразяване на списък (map)

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)`) \rightarrow ?

Изобразяване на списък (`map`) — примери

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

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

Изобразяване на списък (`map`) — примери

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

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

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

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

Изобразяване на списък (`map`) — примери

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

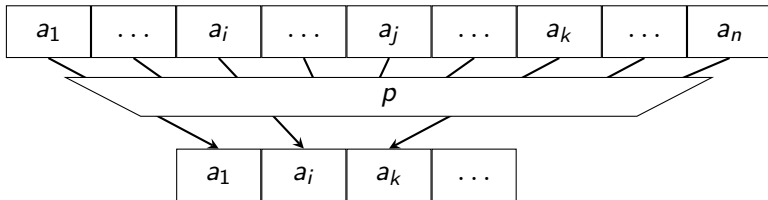
- `(map square '(1 2 3))` \rightarrow `(1 4 9)`
- `(map cadr '((a b c) (d e f) (g h i)))` \rightarrow `(b e h)`
- `(map (lambda (f) (f 2)) (list square 1+ odd?))` \rightarrow `(4 3 #f)`
- `(map (lambda (f) (f 2)) (map twice (list square 1+ boolean?)))` \rightarrow `(16 4 #t)`

Филтриране на списък (filter)

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

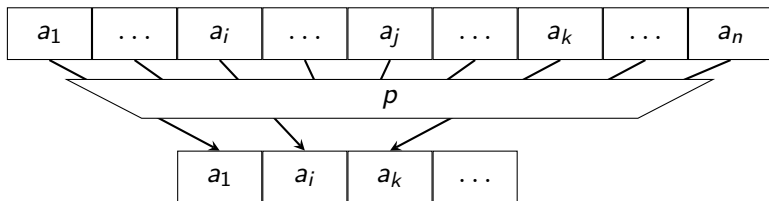
Филтриране на списък (filter)

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)) \rightarrow (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)) \rightarrow (1 3 5)
- (filter pair? '((a b) c () d (e))) \rightarrow ?

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

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)

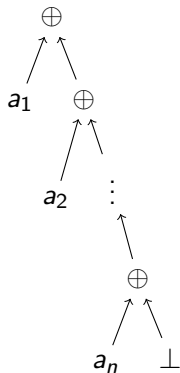
Да се дефинира функция, която по даден списък $l = (a_1 a_2 a_3 \dots a_n)$ пресмята:

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

Дясно свиване (foldr)

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

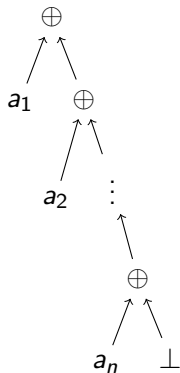
$$a_1 \oplus (a_2 \oplus (\dots \oplus (a_n \oplus \perp) \dots)),$$



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)))))
```

- (foldr * 1 (from-to 1 5)) → ?

Дясно свиване (foldr) — примери

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

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

Дясно свиване (foldr) — примери

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

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

Дясно свиване (foldr) — примери

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

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

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

Дясно свиване (foldr) — примери

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

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

- (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)) → ?

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

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

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

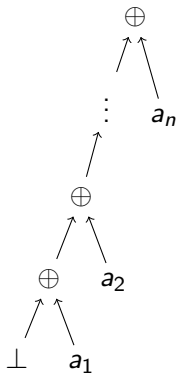
Да се дефинира функция, която по даден списък $l = (a_1 a_2 a_3 \dots a_n)$ пресмята:

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

Ляво свиване (foldl)

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

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

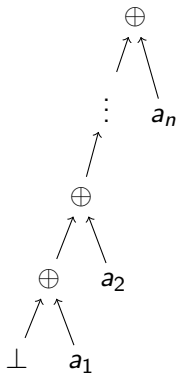


Ляво свиване (foldl)

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

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

```
(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)) \rightarrow 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))` \rightarrow 120
- `(foldl cons '() '(1 5 10))` \rightarrow `(((() . 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))` \rightarrow 120
- `(foldl cons '() '(1 5 10))` \rightarrow `(((() . 1) . 5) . 10)`
- `(foldl ?'() '(1 5 10))` \rightarrow `(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))` \rightarrow 120
- `(foldl cons '() '(1 5 10))` \rightarrow `(((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` \rightarrow `(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))` \rightarrow 120
- `(foldl cons '() '(1 5 10))` \rightarrow `(((() . 1) . 5) . 10)`
- `(foldl (lambda (x y) (cons y x)) '() '(1 5 10))` \rightarrow `(10 5 1)`
- `(foldl list '() '(1 5 10))` \rightarrow ?

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

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

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

foldl в Racket

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

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

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

Задача. Да се намери максималният елемент на списък.

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.
(define (maximum l) (foldr max ? l))

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

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

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

Задача. Да се намери максималният елемент на списък.

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

Можем ли да пропуснем нулевата стойност за непразен списък?

Свиване на непразен списък (foldr1, foldl1)

Задача. Да се намери максималният елемент на списък.

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

Можем ли да пропуснем нулевата стойност за непразен списък?

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

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)))
```