Съставни данни

Наредени двойки и Списъци

Наредена двойка

`(1 . 2)`

- Как се строи?

```
(define x (cons 1 2))
(define x '(1 . 2))
```

`cons` приема две стойности или указатели към стойности

- Как се взима левият елемент?

```
(car x) ; 1 ; Contents of Address Register
```

- Как се взима десният елемент?

```
(cdr x) ; 2 ; Contents of Data Register
```

Влагане на наредени двойки

```
(define x
  (cons 1
        (cons 2
              (cons (cons 3
                          4)
                    5))))

; (1 2 (3 . 4) . 5)
```

И навигиране в тях

```
(define (cddr x) (cdr (cdr x)))
(define (cadr x) (car (cdr x)))
(define (cadddr x) (car (cdr (cdr (cdr x)))))
```

Навигиране във вложени наредени двойки

```
(define x
  (cons 1
        (cons 2
              (cons (cons 3
                          4)
                    5))))

; (1 2 (3 . 4) . 5)
```

```
(cdr x) ; ?
```

```
(cddr x) ; ?
```

```
(caddr x) ; ?
```

```
(caaddr x) ; ?
```

Навигиране във вложени наредени двойки

```
(define x
  (cons 1
        (cons 2
              (cons (cons 3
                          4)
                    5))))

; (1 2 (3 . 4) . 5)
```

```
(cdr x) ; (2 (3 . 4) . 5)
```

```
(cddr x) ; ((3 . 4) . 5)
```

```
(caddr x) ; (3 . 4)
```

```
(caaddr x) ; 2
```

Еквивалентност в съставни данни

```
(eq? 1 1) ; #t
(eq? '(1 . 2) '(1 . 2)) ; #f

(eqv? 1 1) ; #t
(eqv? '(1 . 2) '(1 . 2)) ; #f

(equal? 1 1) ; #t
(equal? '(1 . 2) '(1 . 2)) ; #t

(define x '(1 2 (3 . 4)))

(equal? '(1 2 (3 . 4)) '(1 2 (3 . 4))) ; #t
(equal? x '(1 2 (3 . 4))) ; #t
(equal? '(1 2 (3 . 4)) '(1 2 3 (3 . 4))) ; #f
(eq? x x) ; #t
```

Списъци?

```
(define l
  (cons 1
        (cons 2
              (cons 3
                    4))))
; '(1 2 3 . 4)
```

Списъци!

```
(define l
  (cons 1
        (cons 2
              (cons 3
                    (cons 4 '())))))
; '(1 2 3 4)
```

Списъци

- Празният списък `'()`
- `(cons x l)` добавя отпред `x` като глава на списък `l`
- `(car l)` дава главата на списък `l`
- `(cdr l)` дава опашката на списък `l`
- `(null? l)` проверява дали `l` е `'()`
- Конструираме списъци по 3 начина
  - `(cons 1 (cons 2 (cons 3 (cons 4 '()))))`
  - `(list 1 2 3 4)`
  - `'(1 2 3 4)`
    - За вложени списъци не добавяме `'`:

      good `'(1 2 '(3 4))` -> `(1 2 '(3 4))`

      bad `'(1 2 (3 4))` -> `(1 2 (3 4))`

Линейна рекурсия със списъци

Един от най-често срещаните сценарии:

- Дъно
  - Използваме `null?`, за да проверим дали сме обходили целия списък
  - Или `pair?`, за да проверим, че има точно 1 елемент
- Рекурсивна стъпка
  - Обработваме главата на списъка `(car l)`
  - Подаваме опашката `(cdr l)` на рекурсивната функция
  - Връщаме нов списък чрез `cons`, който обединява новата **глава** и новата **опашка**
    - Защото не можем да прилагаме странични ефекти

map

Да се напише функция `(map f l)`, която трансформира всеки елемент на `l` чрез прилагането на функция `f`.

Примери:

```
(map (lambda (x) (* x x)) '(1 2 3 4)) ; '(1 4 9 16)

(map sqrt '(1 4 64 1024)) ; '(1 2 8 32)
```

map

Да се напише функция `(map f l)`, която трансформира всеки елемент на `l` чрез прилагането на функция `f`.

```
(define (map f l)
  (if (стигнахме ли дъното?)
      (атом, който прекратява рекурсията)
      (комбинираме (елемент от списъка)
                   (рекурсивно извикване))
```

map

Да се напише функция `(map f l)`, която трансформира всеки елемент на `l` чрез прилагането на функция `f`.

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

filter

Да се напише функция `(filter p? l)`, която връща списък с всички елементи на `l`, които удовлетворяват предиката `p?`.

Примери:

```
(filter even? '(1 2 3 4 5)) ; '(2 4)
(filter odd? '(1 2 3 4 5)) ; '(1 3 5)
```

fold a.k.a reduce

Да се напише функция `(fold null-value combine l)`, който акумулира всички елементи на списъка `l` в една стойност, чрез бинарната процедура `combine`. За начална стойност на акумулатора се използва `null-value`.

Примери:

```
(fold 0 + '(1 2 3 4)) ; 10
(fold 1 * '(1 2 3 4)) ; 24
```

Тази процедура е като `accumulate` за редица от цели числа, но вече върху **списък**.