# Структури от данни в Scheme

## асоциативни списъци, дървета, дълбоки списъци

Трифон Трифонов

Функционално програмиране, 2025/26 г.

30 октомври – 6 ноември 2025 г.

# Абстракция със структури от данни

## Дефиниция (Абстракция)

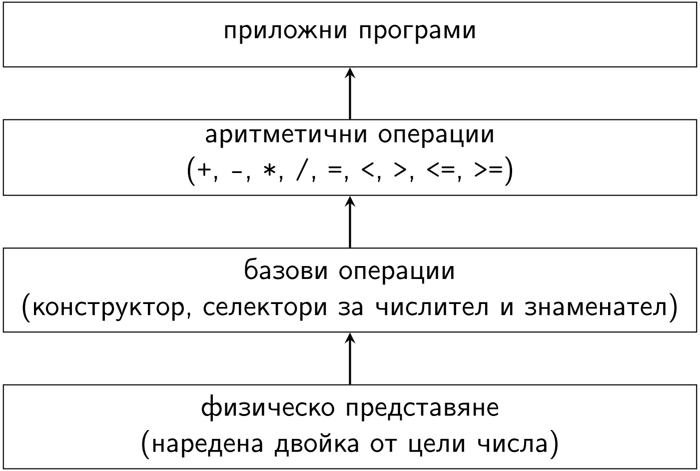
Принцип за разделянето („абстрахирането“) на *представянето* на дадена структура от данни (СД) от нейното *използване*.

- основен принцип на обектно-ориентираното програмиране
- позволява използването на СД преди представянето ѝ да е уточнено
- предимства:
    - програмите работят на по-високо концептуално ниво със СД
    - позволява алтернативни имплементации на дадена СД, подходящи за различни видове задачи
    - влиянието на промени по представянето е ограничено до операциите, които „знаят“ за него
    - подобрения при представянето автоматично се разпространяват до по-горните нива на абстракция

# Пример: рационално число

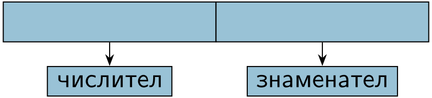
- Логическо описание: обикновена дроб
- Физическо представяне: наредена двойка от цели числа
- Базови операции:
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател
- Аритметични операции:
  - събиране, изваждане
  - умножение, деление
  - сравнение
- Приложни програми

# Нива на абстракция



# Рационални числа

## Физическо представяне




## Базови операции

- `(define make-rat cons)`
- `(define get-numer car)`
- `(define get-denom cdr)`

По-добре:

```
(define (make-rat n d)
  (if (= d 0) (cons n 1) (cons n d)))
```

# Аритметични операции

$$\frac{n_1}{d_1}\frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

```
(define (*rat p q)
  (make-rat
    (* (get-numer p) (get-numer q))
    (* (get-denom p) (get-denom q))))
```

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

```
(define (+rat p q)
  (make-rat
    (+ (* (get-numer p)
          (get-denom q))
       (* (get-denom p)
          (get-numer q)))
    (* (get-denom p) (get-denom q))))
```

$$\frac{n_1}{d_1} < \frac{n_2}{d_2} \leftrightarrow n_1 d_2 < n_2 d_1$$

```
(define (<rat p q)
  (< (* (get-numer p) (get-denom q))
     (* (get-numer q) (get-denom p))))
```

# Програми с рационални числа

$$\sum_{i=0}^{n} \frac{x^i}{i!}$$

```
(define (my-exp x n)
  (accumulate
    +rat (make-rat 0 1) 0 n
    (lambda (i) (make-rat (pow x i) (fact i))) 1+))
```

# Нормализация

**Проблем:** Числителят и знаменателят стават много големи!

**Проблем:** `(<rat (make-rat 1 2) (make-rat 1 -2))` $\longrightarrow$ `#t`

**Идея:** Да работим с *нормализирани* дроби $\frac{p}{q}$, където $p \in \mathbb{Z}, q \in \mathbb{N}^+$ и $gcd(p, q) = 1$.

```scheme
(define (make-rat n d)
  (if (or (= d 0) (= n 0)) (cons 0 1)
    (let* ((g (gcd n d))
           (ng (quotient n g))
           (dg (quotient d g)))
      (if (> dg 0) (cons ng dg)
                   (cons (- ng) (- dg))))))
```

Не е нужно да правим каквито и да е други промени!

# Аритметични операции

$$\frac{n_1}{d_1}\frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

```
(define (*rat p q)
  (make-rat
    (* (get-numer p) (get-numer q))
    (* (get-denom p) (get-denom q))))
```

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

```
(define (+rat p q)
  (make-rat
    (+ (* (get-numer p)
          (get-denom q))
       (* (get-denom p)
          (get-numer q)))
    (* (get-denom p) (get-denom q))))
```

$$\frac{n_1}{d_1} < \frac{n_2}{d_2} \leftrightarrow n_1 d_2 < n_2 d_1$$

```
(define (<rat p q)
  (< (* (get-numer p) (get-denom q))
     (* (get-numer q) (get-denom p))))
```

# Нормализация

**Проблем:** Числителят и знаменателят стават много големи!

**Проблем:** `(<rat (make-rat 1 2) (make-rat 1 -2))` $\longrightarrow$ `#t`

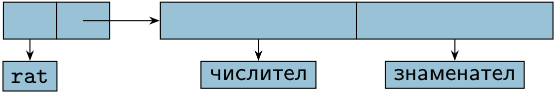
**Идея:** Да работим с *нормализирани* дроби $\frac{p}{q}$, където $p \in \mathbb{Z}, q \in \mathbb{N}^+$ и $gcd(p, q) = 1$.

```
(define (make-rat n d)
  (if (or (= d 0) (= n 0)) (cons 0 1)
    (let* ((g (gcd n d))
           (ng (quotient n g))
           (dg (quotient d g)))
      (if (> dg 0) (cons ng dg)
                   (cons (- ng) (- dg))))))
```

Не е нужно да правим каквито и да е други промени!

# Сигнатура

**Проблем:** Не можем да различим СД с еднакви представяния! (рационално число, комплексно число, точка в равнината)
**Идея:** Да добавим „етикет“ на обекта




```scheme
(define (make-rat n d)
  (cons 'rat
    (if (or (= d 0) (= n 0)) (cons 0 1)
      (let* ((g (gcd n d))
             (ng (quotient n g))
             (dg (quotient d g)))
         (if (> dg 0) (cons ng dg)
                      (cons (- ng) (- dg)))))))
(define get-numer cadr)
(define get-denom cddr)
```

# Проверка за коректност

Вече можем да проверим дали даден обект е рационално число:

```
(define (rat? p)
  (and (pair? p) (eqv? (car p) 'rat)
       (pair? (cdr p))
       (integer? (cadr p)) (positive? (cddr p))
       (= (gcd (cadr p) (cddr p)) 1)))
```

Можем да добавим проверка за коректност:

```
(define (check-rat f)
  (lambda (p)
    (if (rat? p) (f p) 'error)))

(define get-numer (check-rat cadr))
(define get-denom (check-rat cddr))
```

# Капсулация на базови операции

**Проблем:** операциите над СД са видими глобално

**Идея:** да ги направим „private“

```
(define (make-rat n d)
  (lambda (prop)
    (case prop
      ('get-numer n)
      ('get-denom d)
      ('print (cons n d))
      (else 'unknown-prop))))
```

- `(define r (make-rat 3 5))`
- `(r 'get-numer)` $\longrightarrow$ `3`
- `(r 'get-denom)` $\longrightarrow$ `5`
- `(r 'print)` $\longrightarrow$ `(3 . 5)`

# Нормализация при капсулация

```
(define (make-rat n d)
  (let* ((d (if (= 0 d) 1 d))
         (sign (if (> 0 d) 1 -1))
         (g (gcd n d))
         (numer (* sign (quotient n g)))
         (denom (* sign (quotient d g))))
   (lambda (prop)
    (case prop
      ('get-numer numer)
      ('get-denom denom)
      ('print (cons numer denom))
      (else 'unknown-prop)))))
```

- `(define r (make-rat 4 6))`
- `(r 'print)` $\longrightarrow$ `(2 . 3)`

# Капсулация на операции с аргументи

```
(define (make-rat n d)
  (let* ((g (gcd n d))
         (d (if (= 0 d) 1 d))
         (sign (if (> 0 d) 1 -1))
         (numer (* sign (quotient n g)))
         (denom (* sign (quotient d g))))
   (lambda (prop . params)
     (case prop
       ('get-numer numer)
       ('get-denom denom)
       ('print (cons numer denom))
       ('* (let ((r (car params))) (make-rat (* numer (r 'get-numer))
                                             (* denom (r 'get-denom)))))
       (else 'unknown-prop)))))
```

- `(define r1 (make-rat 3 5))`
- `(define r2 (make-rat 5 2))`
- `((r1 '* r2) 'print)` ⟶ `(3 . 2)`

# Извикване на собствени операции

```
(define (make-rat n d)
  (let* ((g (gcd n d))
         (d (if (= 0 d) 1 d))
         (sign (if (> 0 d) 1 -1))
         (numer (* sign (quotient n g)))
         (denom (* sign (quotient d g))))
   (define (self prop . params)
     (case prop
       ('get-numer numer)
       ('get-denom denom)
       ('print (cons numer denom))
       ('* (let ((r (car params)))
             (make-rat (* (self 'get-numer) (r 'get-numer))
                       (* (self 'get-denom) (r 'get-denom)))))
       (else 'unknown-prop)))
   self))
```

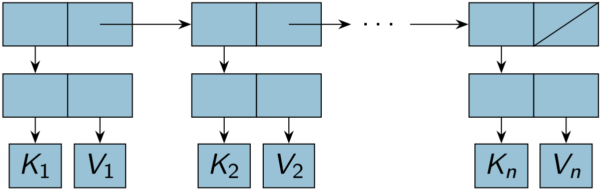
Извикването на метод на обект чрез препратка `self` или `this` се нарича отворена рекурсия.

# Асоциативни списъци

## Дефиниция

Асоциативните списъци (още: речник, хеш, map) са списъци от наредени двойки (<ключ> . <стойност>). <ключ> и <стойност> може да са произволни S-изрази.

$((K_1 \ . \ V_1) \ (K_2 \ . \ V_2) \ \ldots (K_n \ . \ V_n))$



# Примери за асоциативни списъци

- `((1 . 2) (2 . 3) (3 . 4))`
- `((a . 10) (b . 12) (c . 18))`
- `((l1 1 8) (l2 10 1 2) (l3))`
- `((al1 (1 . 2) (2 . 3)) (al2 (b)) (al3 (a . b) (c . d)))`

**Пример:** Създаване на асоциативен списък по списък от ключове и функция:

```scheme
(define (make-alist f keys)
  (map (lambda (x) (cons x (f x))) keys))
```

`(make-alist square '(1 3 5))` $\longrightarrow$ `((1 . 1) (3 . 9) (5 . 25))`

# Селектори за асоциативни списъци

- `(define (keys alist) (map car alist))`
- `(define (values alist) (map cdr alist))`
- (assoc <ключ> <асоциативен-списък>)
    - Ако <ключ> се среща сред ключовете на <асоциативен-списък>, връща първата двойка (<ключ> . <стойност>)
    - Ако <ключ> не се среща сред ключовете, връща #f
    - Сравнението се извършва с equal?
- (assv <ключ> <асоциативен-списък>)
    - също като assoc, но сравнява с eqv?
- (assq <ключ> <асоциативен-списък>)
    - също като assoc, но сравнява с eq?

# Трансформации над асоциативни списъци

- Изтриване на ключ и съответната му стойност (ако съществува):

```
(define (del-assoc key alist)
  (filter (lambda (kv) (not (equal? (car kv) key))) alist))
```

- Задаване на стойност за ключ (изтривайки старата, ако има такава):

```
(define (add-assoc key value alist)
  (cons (cons key value) (del-assoc key alist)))
```

# Задачи за съществуване

**Задача.** Да се намери има ли елемент на l, който удовлетворява p.
**Формула:** $\exists x \in l : p(x)$
**Решение:**

```
(define (search p l)
  (and (not (null? l))
       (or (p (car l)) (search p (cdr l)))))
```

Важно свойство: Ако p връща „свидетел“ на истинността на свойството *p* (както например memv или assv), то search също връща този „свидетел“.
**Пример:**

```
(define (assv key al)
  (search (lambda (kv) (and (eqv? (car kv) key) kv)) al))
```

# Задачи за всяко

**Задача.** Всеки елемент на l да се трансформира по дадено правило f.
**Формула:** $\{f(x) \mid x \in l\}$
**Решение:** `(map f l)`

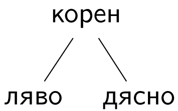
**Задача.** Да се изберат тези елементи от l, които удовлетворяват p.
**Формула:** $\{x \mid x \in l \wedge p(x)\}$
**Решение:** `(filter p l)`

**Задача.** Да се провери дали всички елементи на l удовлетворяват p.
**Формула:** $\forall x \in l : p(x) \leftrightarrow \neg\exists x \in l : \neg p(x)$
**Решение:**

```
(define (all? p? l)
  (not (search (lambda (x) (not (p? x))) l)))
```

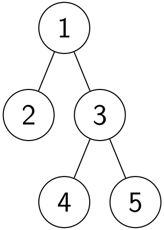
# Представяне на двоични дървета

Представяме двоични дървета като вложени списъци от три елемента:




(<корен> <ляво> <дясно>)

Пример:




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

# Базови операции

Проверка за коректност:

```
(define (tree? t)
  (or (null? t)
      (and (list t) (= (length t) 3))
           (tree? (cadr t))
           (tree? (caddr t))))
```

Конструктори:

```
(define empty-tree '())
(define (make-tree root left right) (list root left right))
```

Селектори:

```
(define root-tree car)
(define left-tree cadr)
(define right-tree caddr)
(define empty-tree? null?)
```

# Разширени операции

Дълбочина на дърво:

```
(define (depth-tree t)
  (if (empty-tree? t) 0
      (1+ (max (depth (left-tree t))
               (depth (right-tree t))))))
```

Намиране на поддърво:

```
(define (memv-tree x t)
  (and (not (empty-tree? t))
       (or (and (eqv? x (root-tree t)) t)
           (memv-tree x (left-tree t))
           (memv-tree x (right-tree t)))))
```

# Търсене на път в двоично дърво

**Задача:** Да се намери в дървото път от корена до даден възел x.

```
(define (path-tree x t)
  (and (not (empty-tree? t))
       (or (and (eqv? x (root-tree t)) (list x))
           (cons#f (root-tree t)
                   (or (path-tree x (left-tree t))
                       (path-tree x (right-tree t)))))))

(define (cons#f h t) (and t (cons h t)))
```

# Работа с дълбоки списъци

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

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

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

За удобство можем да дефинираме функцията `atom?`:

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

# Примери

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

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

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

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

# Примери

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

(deep-reverse '((1 (2)) (((3) 4) (5 (6)) () (7)) 8)) $\longrightarrow$

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

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

# Свиване на дълбоки списъци

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

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

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

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

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

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

# Директна реализация на deep-foldr

Как работи `deep-foldr`?

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

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

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

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