Структури от данни в Scheme

матрици, дървета, асоциативни списъци, графи

Трифон Трифонов

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

10 ноември 2016 г.

Представяне на матрици

Можем да представим матрица като списък от списък от елементи:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

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

Представяне на матрици

Можем да представим матрица като списък от списък от елементи:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

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

Проверка за коректност:

Представяне на матрици

Можем да представим матрица като списък от списък от елементи:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad \texttt{((1 2 3) (4 5 6))}$$

Проверка за коректност:

```
(define (all p? l) (foldr (lambda (x y) (and x y)) #t l))

(define (matrix? m)
  (and (list? m)
       (not (null? (car m)))
       (all list? m)
       (all (lambda (row) (= (length row)
                             (length (car m)))) m)))
```

Базови операции

Брой редове и стълбове

Базови операции

Брой редове и стълбове

```
(define (get-rows m) (length m))
(define (get-columns m) (length (car m)))
```

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Намиране на първи ред и стълб

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Намиране на първи ред и стълб

```
(define (get-first-row m) (car m))
(define (get-first-column m) (map car m))
```

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Намиране на първи ред и стълб

```
(define get-first-row car)
(define (get-first-column m) (map car m))
```

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Намиране на първи ред и стълб

```
(define get-first-row car)
(define (get-first-column m) (map car m))
```

Изтриване на първи ред и стълб

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Намиране на първи ред и стълб

```
(define get-first-row car)
(define (get-first-column m) (map car m))
```

Изтриване на първи ред и стълб

```
(define (del-first-row m) (cdr m))
(define (del-first-column m) (map cdr m))
```

Базови операции

Брой редове и стълбове

```
(define get-rows length)
(define (get-columns m) (length (car m)))
```

Намиране на първи ред и стълб

```
(define get-first-row car)
(define (get-first-column m) (map car m))
```

Изтриване на първи ред и стълб

```
(define del-first-row cdr)
(define (del-first-column m) (map cdr m))
```

Разширени операции

Намиране на ред и стълб по индекс

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate ? ? ? ?
              ? ?))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate cons ? ? ?
              ? ?))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate cons '() ? ?
              ? ?))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate cons '() 0 ?
              ? ?))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate cons '() 0 (- (get-columns m) 1)
              ? ?))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate cons '() 0 (- (get-columns m) 1)
              (lambda (i) (get-column i m)) ?))
```

Разширени операции

Намиране на ред и стълб по индекс

```
(define (get-row i m) (list-ref m i))
(define (get-column i m)
  (map (lambda (row) (list-ref row i)) m))
```

Транспониране

```
(define (transpose m)
  (accumulate cons '() 0 (- (get-columns m) 1)
              (lambda (i) (get-column i m)) 1+))
```

Аритметични операции

Събиране на матрици

Аритметични операции

Събиране на матрици

```
(define (sum-vectors v1 v2) (map + v1 v2))
(define (sum-matrices m1 m2) (map sum-vectors m1 m2))
```

Аритметични операции

Събиране на матрици

```
(define (sum-vectors v1 v2) (map + v1 v2))
(define (sum-matrices m1 m2) (map sum-vectors m1 m2))
```

Умножение на матрици

Аритметични операции

Събиране на матрици

```
(define (sum-vectors v1 v2) (map + v1 v2))
(define (sum-matrices m1 m2) (map sum-vectors m1 m2))
```

Умножение на матрици ($c_{i,j} = \vec{a}_i \cdot \vec{b}_j^T = \sum_{k=0}^{n} A_{i,k} B_{k,j}$)

Аритметични операции

Събиране на матрици

```
(define (sum-vectors v1 v2) (map + v1 v2))
(define (sum-matrices m1 m2) (map sum-vectors m1 m2))
```

Умножение на матрици ($c_{i,j} = \vec{a}_i \cdot \vec{b}_j^T = \sum_{k=0}^{n} A_{i,k} B_{k,j}$)

```
(define (mult-vectors v1 v2) (apply + (map * v1 v2)))
(define (mult-matrices m1 m2)
  (let ((m2t (transpose m2)))
       (map (lambda (row)
              (map (lambda (column) (mult-vectors row column))
                   m2t))
            m1)))
```

Абстракция със структури от данни

Дефиниция (Абстракция)

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

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

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

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:

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

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

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

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

Пример: рационално число

- Логическо описание: обикновена дроб
- Физическо представяне: точкова двойка от цели числа

Пример: рационално число

- Логическо описание: обикновена дроб
- Физическо представяне: точкова двойка от цели числа
- Базови операции:
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател

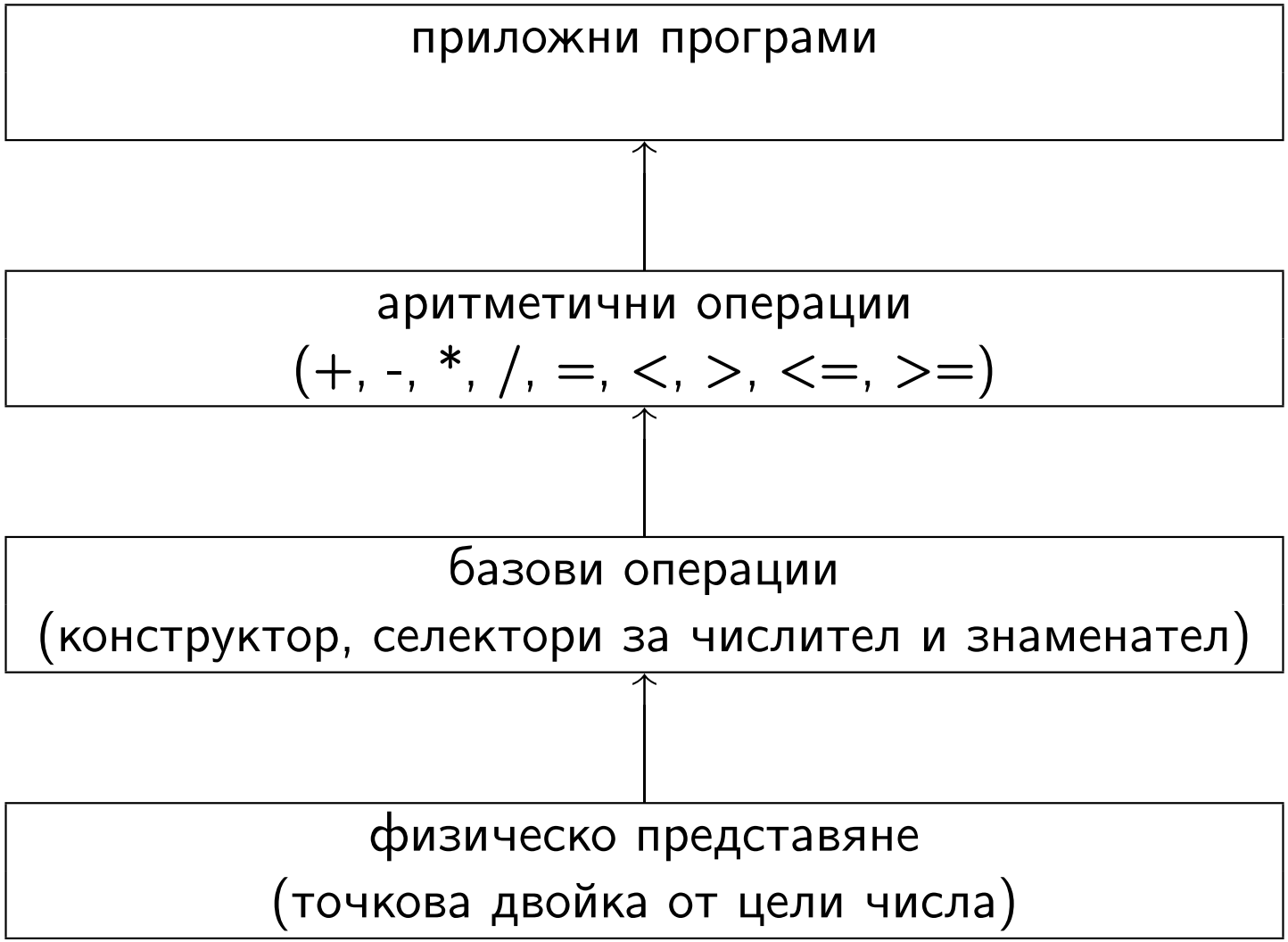
Пример: рационално число

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

Пример: рационално число

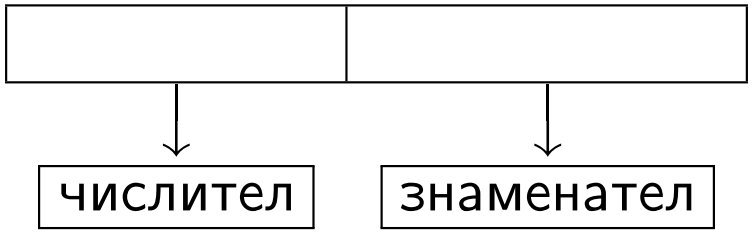
- Логическо описание: обикновена дроб
- Физическо представяне: точкова двойка от цели числа
- Базови операции:
  - конструиране на рационално число
  - получаване на числител
  - получаване на знаменател
- Аритметични операции:
  - събиране, изваждане
  - умножение, деление
  - сравнение
- Приложни програми

Нива на абстракция



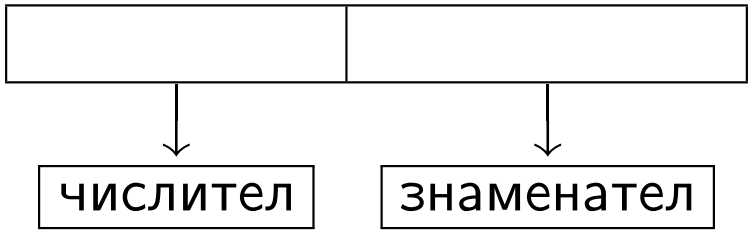
Рационални числа

Физическо представяне



Рационални числа

Физическо представяне

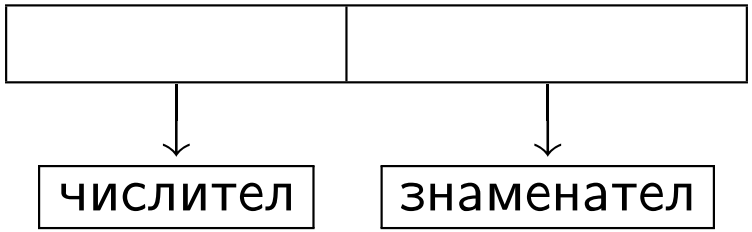


Базови операции

- `(define (make-rat n d) (cons n d))`

Рационални числа

Физическо представяне

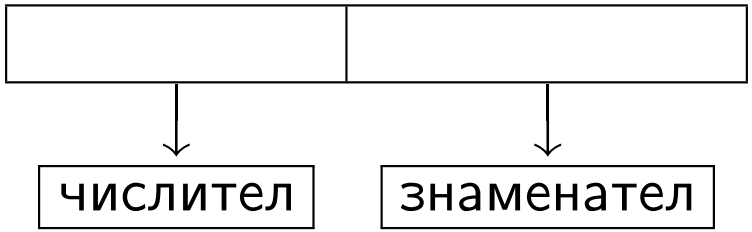


Базови операции

- `(define make-rat cons)`

Рационални числа

Физическо представяне

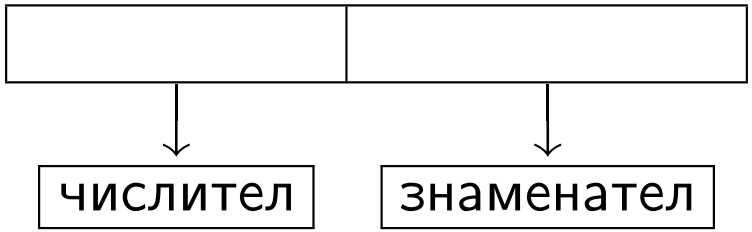


Базови операции

- `(define make-rat cons)`
- `(define (get-numer r) (car r))`

Рационални числа

Физическо представяне

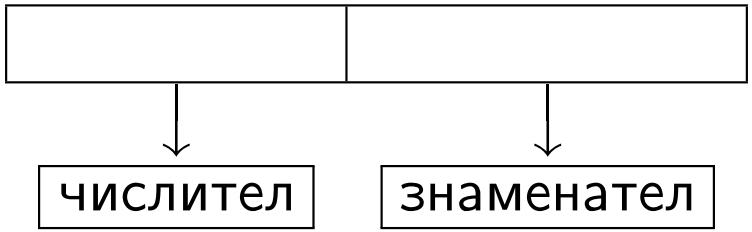


Базови операции

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

Рационални числа

Физическо представяне

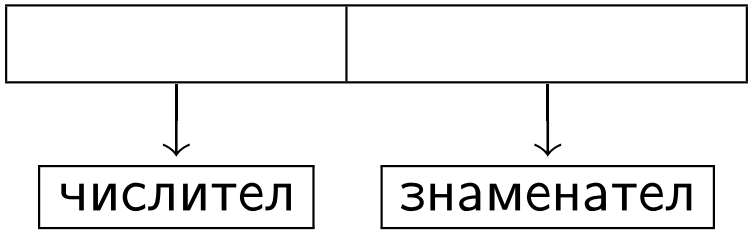


Базови операции

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

Рационални числа

Физическо представяне

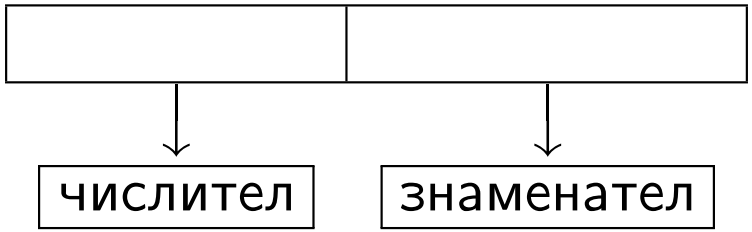


Базови операции

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

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))))
```

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))))
```

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 ? ? 0 n
              ? 1+))
```

Програми с рационални числа

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

```
(define (my-exp x n)
  (accumulate +rat ? 0 n
              ? 1+))
```

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
              ? 1+))
```

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!

Нормализация

Проблем: Числителят и знаменателят стават много големи!

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

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 $gcd(p, q) = 1$.

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))))))
```

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))))))
```

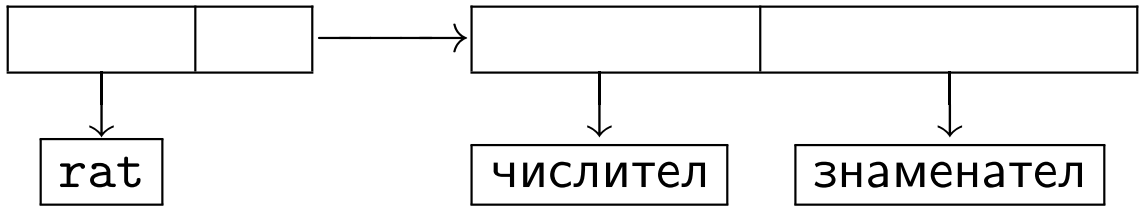
Не е нужно да правим каквито и да е други промени!

Сигнатура

Проблем: Не можем да различим СД с еднакви представяния!
(рационално число, комплексно число, точка в равнината)

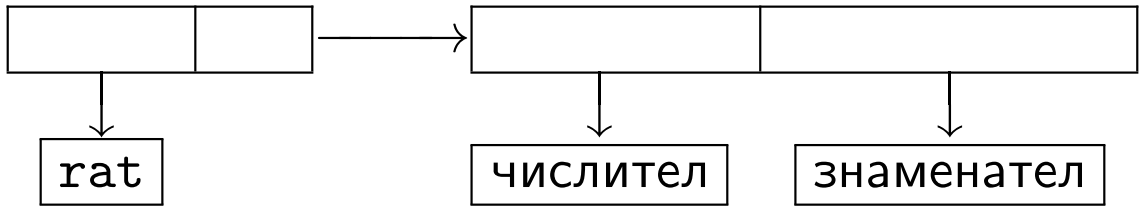
Сигнатура

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



Сигнатура

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




```
(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) (eq? (car p) 'rat)
       (pair? (cdr p))
       (integer? (cadr p)) (integer? (cddr p))))
```

Проверка за коректност

Вече можем да проверим дали даден обект е рационално число:

```
(define (rat? p)
  (and (pair? p) (eq? (car p) 'rat)
       (pair? (cdr p))
       (integer? (cadr p)) (integer? (cddr p))))
```

Можем да добавим проверка за коретност:

```
(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

Капсулация на базови операции

Проблем: операциите над СД са видими глобално

Идея: да ги направим “private”

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)))))
```

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)))))
```

- `(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* ((g (gcd n d))
         (numer (quotient n g))
         (denom (quotient d g)))
    (lambda (prop)
     (case prop
       ('get-numer numer)
       ('get-denom denom)
       ('print (cons numer denom))))))
```

Нормализация при капсулация

```
(define (make-rat n d)
  (let* ((g (gcd n d))
         (numer (quotient n g))
         (denom (quotient d g)))
    (lambda (prop)
      (case prop
        ('get-numer numer)
        ('get-denom denom)
        ('print (cons numer denom))))))
```

- `(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))
         (numer (quotient n g))
         (denom (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))))))))))
```

Капсулация на операции с аргументи

```
(define (make-rat n d)
  (let* ((g (gcd n d))
         (numer (quotient n g))
         (denom (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)))))))))
```

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

Извикване на собствени операции

```
(define (make-rat n d)
  (let* ((g (gcd n d))
         (numer (quotient n g))
         (denom (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))))))
    self))
```

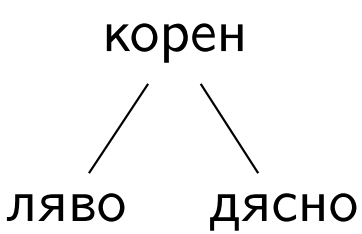
Извикване на собствени операции

```
(define (make-rat n d)
  (let* ((g (gcd n d))
         (numer (quotient n g))
         (denom (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))))))
    self))
```

Извикването на метод на обект чрез референция към себе си `self` или `this` се нарича отворена рекурсия.

Представяне на двоични дървета

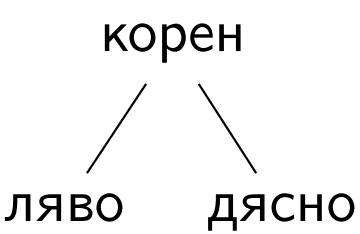
Представяме двоични дървета като вложени списъци от три елемента:



(<корен> <ляво> <дясно>)

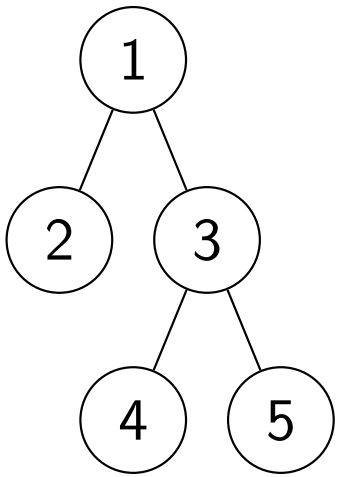
Представяне на двоични дървета

Представяме двоични дървета като вложени списъци от три елемента:



(<корен> <ляво> <дясно>)

Пример:




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

Базови операции

Проверка за коректност:

Базови операции

Проверка за коректност:

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

Базови операции

Проверка за коректност:

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

Конструктори:

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))
```

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:

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?)
```