

Структури от данни в Scheme

матрици, дървета, асоциативни списъци, графи

Трифон Трифонов

Функционално програмиране, 2017/18 г.

16–23 ноември 2017 г.

Представяне на матрици

Можем да представим матрица като списък от списък от елементи:

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

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

```
(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

```
(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

```
(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*.

- 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-numer q)
                  (get-denom p)))
            (* (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)) → #t`

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

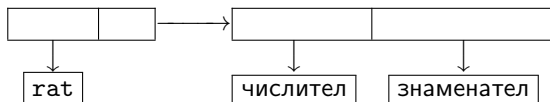
```
(define (make-rat n d)
  (if (or (= d 0) (= n 0)) (cons 0 1)
      (let* ((g (gcd n d))
              (ng (quotient n g))
              (dg (quotient d g)))
        (if (> dg 0) (cons ng dg)
            (cons (- ng) (- dg))))))
```

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



```
(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 caddr)
```

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 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))))))
```

- `(define r (make-rat 3 5))`
- `(r 'get-numer) → 3`
- `(r 'get-denom) → 5`
- `(r 'print) → (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))))))
```

- (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))))))))
```

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

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

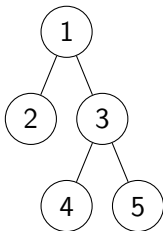
Представяне на двоични дървета

Представяме двоични дървета като вложени списъци от три елемента:



(<корен> <ляво> <дясно>)

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 (memq-tree x t)
  (cond ((empty-tree? t) #f)
        ((eq? x (root-tree t)) t)
        (else (or (memq-tree x (left-tree t))
                    (memq-tree x (right-tree t))))))
```

Търсене на път в двоично дърво

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

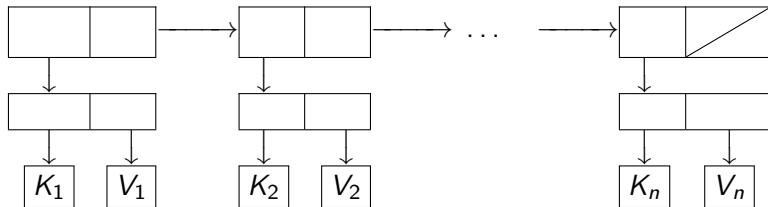
```
(define (path-tree x t)
  (cond ((empty-tree? t) #f)
        ((eq? x (root-tree t)) (list x))
        (else (cons#f (root-tree t)
                        (or (path-tree x (left-tree t))
                            (path-tree x (right-tree t)))))))

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

Асоциативни списъци

Дефиниция

Асоциативните списъци (още: речник, хеш, map) са списъци от наредени двойки ($\langle \text{ключ} \rangle . \langle \text{стойност} \rangle$). $\langle \text{ключ} \rangle$ и $\langle \text{стойност} \rangle$ може да са произволни S-изрази.

$$((K_1 . V_1) (K_1 . V_2) \dots (K_n . V_n))$$


Примери за асоциативни списъци

- `((1 . 2) (2 . 3) (3 . 4))`
- `((a . 10) (b . 12) (c . 18))`
- `((11 1 8) (12 10 1 2) (13))`
- `((a11 (1 . 2) (2 . 3)) (a12 (b)) (a13 (a . b) (c . d)))`

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

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

```
(make-alist square '(1 3 5)) → ((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 `memq` или `assq`), то `search` също връща този “свидетел”.

Пример:

```
(define (assq key al)
  (search (lambda (kv) (and (eq? (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)`

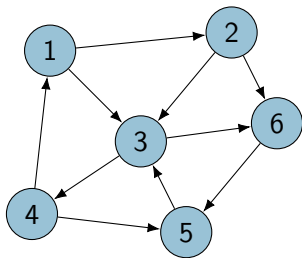
Задача. Да се провери дали всички елементи на l удовлетворяват p .

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 графи чрез асоциативни списъци



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

Асоциативен списък, в който **ключовете** са върховете, а **стойностите** са списъци от техните деца.

Абстракция за граф

```
(define (vertices g) (map car g))
```

```
(define (children v g)
  (cdr (assq v g)))
```

$$\{u \mid u \leftarrow v\}$$

```
(define (edge? u v g)
  (memq v (children u g)))
```

$$u \xrightarrow{?} v$$

```
(define (map-children v f g)
  (map f (children v g)))
```

$$\forall u \leftarrow v$$

```
(define (search-child v f g)
  (search f (children v g)))
```

$$\exists u \leftarrow v$$

Абстракция за граф

Абстракция чрез капсулация

```

(define (make-graph g)
  (define (self prop . params)
    (case prop
      ('print g)
      ('vertices (map car g))
      ('children (let ((v (car params)))
                   (cdr (assq v g))))
      ('edge? (let ((u (car params)) (v (cadr params)))
                 (memq v (self 'children u))))
      ('map-children (let ((v (car params))
                           (f (cadr params)))
                       (map f (self 'children v))))
      ('search-child (let ((v (car params))
                           (f (cadr params)))
                       (exists u (self 'children v)
                                (lambda ()
                                  (f u (cadr params)))))))
    self)

```

$\{u \mid u \leftarrow v\}$
 $u \xrightarrow{?} v$
 $\forall u \leftarrow v$
 $\exists u \leftarrow v$

Локални задачи

Задача. Да се намерят върховете, които нямат деца.

Решение. $\text{childless}(g) = \{v \mid \nexists u \leftarrow v\}$

```
(define (childless g)
  (filter (lambda (v) (null? (children v g))) (vertices g)))
```

Задача. Да се намерят родителите на даден връх.

Решение. $\text{parents}(v, g) = \{u \mid u \rightarrow v\}$

```
(define (parents v g)
  (filter (lambda (u) (edge? u v g)) (vertices g)))
```


Проверка за симетричност

Задача. Да се провери дали граф е симетричен.

Решение. $\text{symmetric?}(g) = \forall u \forall v \leftarrow u : v \rightarrow u$

```
(define (symmetric? g)
  (all (lambda (u)
        (all (lambda (v) (edge? v u g))
              (children u g)))
        (vertices g)))
```

Схема на обхождане в дълбочина

Обхождане на връх v :

- Обходи последователно всички наследници на v

(`define` (`dfs` `u` `g`)

(`<функция-за-обработка>` (`lambda` (`v`) (`<действие>` (`dfs` `v` `g`)))
(`children` `u` `g`)))

- **Имаме ли дъно?**

- Да: при празен списък от наследници няма рекурсивно извикване!

- **Какво се случва ако графът е цикличен?**

- Програмата също зацикля! Как да се справим с този проблем?
- Трябва да помним през кои върхове сме минали!
- Два варианта:
 - 1 да помним всички обходени до момента върхове
 - 2 да помним текущия път

Търсене на път в дълбочина

Задача. Да се намери път от u до v , ако такъв има.

Решение. Има път от u до v , ако:

- $u = v$, или
- има дете $w \leftarrow u$, така че има път от w до v

```
(define (dfs-path u v g)
  (define (dfs-search path)
    (let ((current (car path)))
      (cond ((eq? current v) (reverse path))
            ((memq current (cdr path)) #f)
            (else (search-child current
                                   (lambda (w) (dfs-search (cons w path))))
              g))))
  (dfs-search (list u)))
```

Директно рекурсивно решение, работи само за ацикличен граф!

Итеративното натрупване на пътя позволява да правим проверки за цикъл.

Схема на обхождане в ширина

Обхождане, започващо от връх u :

- Маркира се u за обхождане на ниво 1
- За всеки връх v избран за обхождане на ниво n :
 - Маркират се всички деца s на v за обхождане на ниво $n + 1$

```
(define (bfs u g)
  (define (bfs-level l)
    (if (null? l) <дъно>
        (bfs-level
         (<функция-за-обработка> (lambda (v) (children v g))
                                  1))))
  (bfs-level (list u)))
```

- **Какво се случва ако графът е цикличен?**
 - Ако има път: намира го.
 - Ако няма път: програмата зацикля! Как да се справим с този проблем?
 - Трябва да помним през кои върхове сме минали!
 - Нивото трябва да представлява **списък от пътища**

Разширяване на пътища

Удобно е пътищата да са представени като **стек**

- последно посещеният възел е най-лесно достъпен

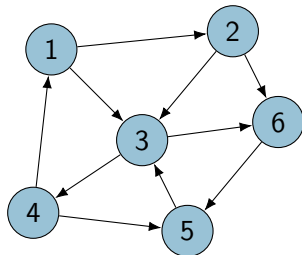
```
(extend '(2 1)) → ((3 2 1) (6 2 1))
```

```
(define (extend path)
  (map-children (car path)
    (lambda (u) (cons u path)) g))
```

Трябва да филтрираме циклите:

```
(define (acyclic? path)
  (not (memq (car path) (cdr path))))
```

```
(define (extend-acyclic path)
  (filter acyclic? (extend path)))
```



Търсене на път в ширина

Задача. Да се намери **най-краткия** път от u до v , ако такъв има.

Решение. Обхождаме в ширина от u докато намерим ниво, в което има път, завършващ във върха v .

```
(define (bfs-path u v g)
  (define (extend path) ...)
  (define (extend-acyclic path) ...)
  (define (extend-level level)
    (apply append (map extend-acyclic level)))
  (define (target-path path)
    (and (eq? (car path) v) path))

  (define (bfs-level level)
    (and (not (null? level))
         (or (search target-path level)
              (bfs-level (extend-level level)))))

  (bfs-level (list (list u))))
```