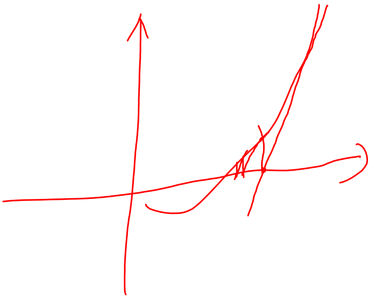
Структури от данни в Scheme

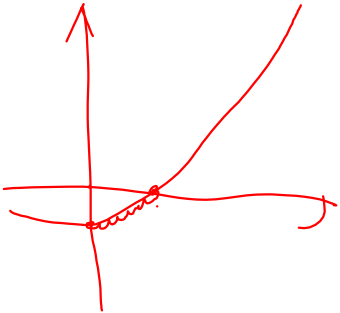
матрици, дървета, асоциативни списъци, графи

Трифон Трифонов

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

10–17 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:

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

```
((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 (g x) (f x))

```
(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+))
```

(define (transpose m) (apply map list m))

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

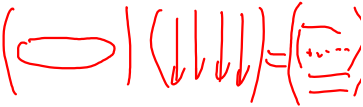
Аритметични операции

Събиране на матрици

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

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

```scheme
(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

приложни програми

↑

аритметични операции
(+, -, *, /, =, <, >, <=, >=)

↑

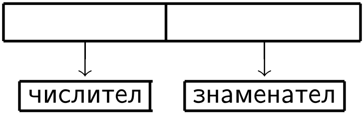
базови операции
(конструктор, селектори за числител и знаменател)

↑

физическо представяне
(точкова двойка от цели числа)

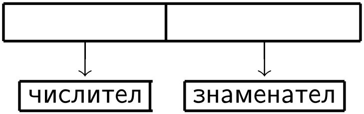
Рационални числа

Физическо представяне



Рационални числа

Физическо представяне

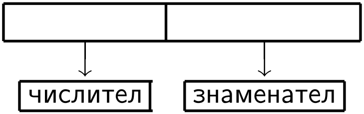


Базови операции

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

Рационални числа

Физическо представяне

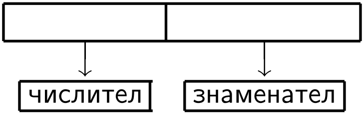


Базови операции

- `(define make-rat cons)`

Рационални числа

Физическо представяне

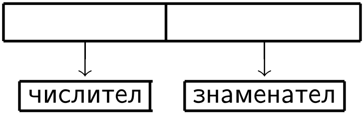


Базови операции

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

Рационални числа

Физическо представяне

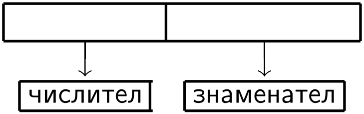


Базови операции

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

Рационални числа

Физическо представяне

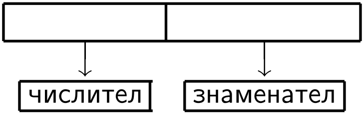


Базови операции

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

Рационални числа

Физическо представяне

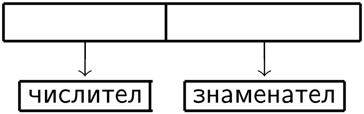


Базови операции

- `(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

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

```scheme
(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 $gcd(p, q) = 1$.

```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))))))
```

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))))))
```

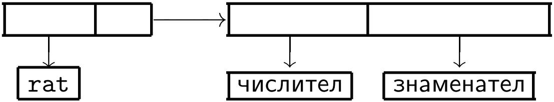
Не е нужно да правим каквито и да е други промени!

Сигнатура

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

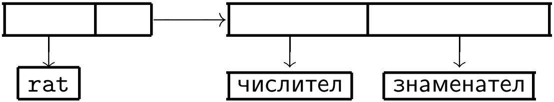
Сигнатура

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

```scheme
(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)` ⟶ `(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:

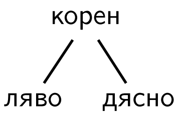
корен

ляво дясно

(<корен> <ляво> <дясно>)

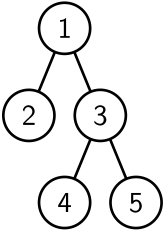
Представяне на двоични дървета

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?)
```

Разширени операции

Дълбочина на дърво:

Разширени операции

Дълбочина на дърво:

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

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:

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.

Търсене на път в двоично дърво

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

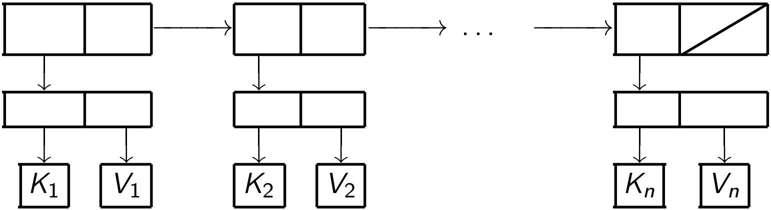
```
(define (path-tree x t)
  (cond ((empty-tree? t) #f)
        ((eq? x (root-tree t)) (list x))
        (else (cons#f (root-tree t)
                      (or (path-tree x (left-tree t))
                          (path-tree x (right-tree t)))))))

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

Асоциативни списъци

Дефиниция

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

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



Примери за асоциативни списъци

- `((1 . 2) (2 . 3) (3 . 4))`

Примери за асоциативни списъци

- `((1 . 2) (2 . 3) (3 . 4))`
- `((a . 10) (b . 12) (c . 18))`

Примери за асоциативни списъци

- ((1 . 2) (2 . 3) (3 . 4))
- ((a . 10) (b . 12) (c . 18))
- ((l1 1 8) (l2 10 1 2) (l3))

(l1 . (1 8)) . (l2 . (10 1 2)) (l3 . ())

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)))

(b . ())

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 асоциатвен списък по списък от ключове и функция:

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

keys

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 асоциатвен списък по списък от ключове и функция:

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

`(make-alist square '(1 3 5))` ⟶ `((1 . 1) (3 . 9) (5 . 25))`

Операции над асоциативни списъци

- `(define (keys alist) (map car alist))`

Операции над асоциативни списъци

- `(define (keys alist) (map car alist))`
- `(define (values alist) (map cdr alist))`

Операции над 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?`

Операции над 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?`
- `(assqv <ключ> <асоциативен-списък>)`
  - също като `assoc`, но сравнява с `eqv?`

Операции над 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?`
- `(assqv <ключ> <асоциативен-списък>)`
  - също като `assoc`, но сравнява с `eqv?`
- `(assq <ключ> <асоциативен-списък>)`
  - също като `assoc`, но сравнява с `eq?`

Операции над 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?
- `(assqv <ключ> <асоциативен-списък>)`
    - също като assoc, но сравнява с eqv?
- `(assq <ключ> <асоциативен-списък>)`
    - също като assoc, но сравнява с eq?
- Изтриване на ключ и съответната му стойност:

Операции над 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?
- `(assqv <ключ> <асоциативен-списък>)`
  - също като assoc, но сравнява с eqv?
- `(assq <ключ> <асоциативен-списък>)`
  - също като assoc, но сравнява с eq?
- Изтриване на ключ и съответната му стойност:

```
(define (del-key-value key alist)
  (filter (lambda (kv) (not (eq? (car kv) key))) alist))
```

Задаване на стойност за ключ

Вариант №1 (грозен и по-бърз):

```
(define (add-key-value key value alist)
   (let ((new-kv (cons key value)))
        (cond ((null? alist) (list new-kv))
              ((eq? (caar alist) key)
               (cons new-kv (cdr alist)))
              (else (cons (car alist)
                    (add-key-value key value (cdr alist)))))))
```

Задаване на стойност за ключ

Вариант №1 (грозен и по-бърз):

```
(define (add-key-value key value alist)
   (let ((new-kv (cons key value)))
        (cond ((null? alist) (list new-kv))
              ((eq? (caar alist) key)
               (cons new-kv (cdr alist)))
              (else (cons (car alist)
                    (add-key-value key value (cdr alist)))))))
```

Вариант №2 (красив и по-бавен):

```
(define (add-key-value key value alist)
  (let ((new-kv (cons key value)))
       (if (assq key alist)
           (map (lambda (kv) (if (eq? (car kv) key)
                                 new-kv kv)) alist)
           (cons new-kv alist))))
```

Задачи за съществуване

Задача. Да се намери има ли елемент на l, който удовлетворява p.

Задачи за съществуване

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

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)))))
```

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".

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 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.

Задачи за всяко

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

Задачи за всяко

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

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.

Задачи за всяко

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

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

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)`

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.

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: $\forall x \in l : p(x)$

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: $\forall x \in l : p(x) \leftrightarrow \neg\exists x \in l : \neg p(x)$

Задачи за всяко

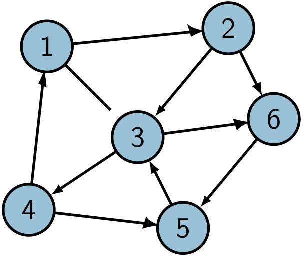
Задача. Всеки елемент на l да се трансформира по дадено правило f.
Формула: $\{f(x) \mid x \in l\}$
Решение: `(map f l)`

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 (forall p l)
  (not (search (lambda (x) (not (p x))) l)))
```

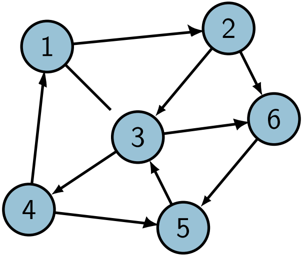
Представяне на графи чрез асоциативни списъци




```
((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.

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

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

(define (children v g)                    ;; {u|u ← v}
  (cdr (assq v g)))

(define (edge? u v g)                     ;; u -?-> v
  (memq v (children u g)))

(define (map-children v f g)              ;; ∀u ← v
  (map f (children v g)))

(define (search-child v f g)              ;; ∃u ← v
  (search f (children v g)))
```

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))))                ;; {u|u ← v}
      ('edge? (let ((u (car params)) (v (cadr params)))
                (memq v (self 'children u))))       ;; u →? v
      ('map-children (let ((v (car params))
                           (f (cadr params)))       ;; ∀u ← v
                       (map f (self 'children v))))
      ('search-child (let ((v (car params))
                           (f (cadr params)))       ;; ∃u ← v
                       (search f (self 'children v))))))
  self)
```

Локални задачи

Задача. Да се намерят върховете, които нямат деца.

Локални задачи

Задача. Да се намерят върховете, които нямат деца.
Решение. `childless(g)` $= \{v | \nexists u \leftarrow v\}$

Локални задачи

Задача. Да се намерят върховете, които нямат деца.
Решение. $\texttt{childless(g)} = \{v | \nexists u \leftarrow v\}$

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

Локални задачи

Задача. Да се намерят върховете, които нямат деца.
Решение. childless(g) $= \{v | \nexists u \leftarrow v\}$

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

Задача. Да се намерят родителите на даден връх.

Локални задачи

Задача. Да се намерят върховете, които нямат деца.
Решение. childless(g) $= \{v | \nexists u \leftarrow v\}$

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

Задача. Да се намерят родителите на даден връх.
Решение. parents(v, g) $= \{u | u \rightarrow v\}$

Локални задачи

Задача. Да се намерят върховете, които нямат деца.
Решение. childless(g) $= \{v | \nexists u \leftarrow v\}$

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

Задача. Да се намерят родителите на даден връх.
Решение. parents(v, g) $= \{u | u \rightarrow v\}$

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

Проверка за симетричност

Задача. Да се провери дали граф е симетричен.

Проверка за симетричност

Задача. Да се провери дали граф е симетричен.

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

Проверка за симетричност

Задача. Да се провери дали граф е симетричен.
Решение. symmetric?(g) $= \forall u \forall v \leftarrow u : v \rightarrow u$

```
(define (symmetric? g)
  (forall (lambda (u)
            (forall (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

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)))
```

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?

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!

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?

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?

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!

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:

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

Схема на обхождане в дълбочина

Обхождане на връх v:

- Обходи последователно всички наследници на v

```scheme
(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

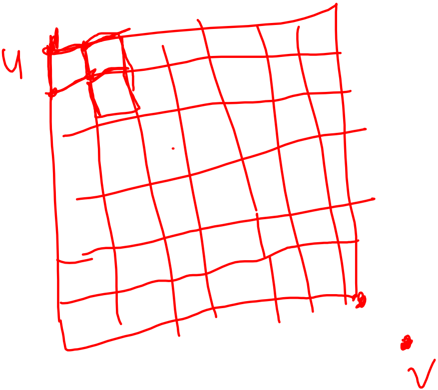

u
v

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.

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:

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

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)
  (if (eq? u v) (list u)
      (search-child u (lambda (c)
                           (cons#f u (dfs-path c v g))) g)))
```

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 ← u, така че има път от w до v

```
(define (dfs-path u v g)
  (if (eq? u v) (list u)
      (search-child u (lambda (c)
                               (cons#f u (dfs-path c v g))) g)))
```

Директно рекурсивно решение, работи само за ацикличен граф!

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 ← u, така че има път от w до v

```
(define (dfs-path u v g)
  (if (eq? u v) (list u)
      (search-child u (lambda (c)
                         (cons#f u (dfs-path c v g))) g)))
```

Директно рекурсивно решение, работи само за ацикличен граф!

Итеративното натрупване на пътя позволява да правим проверки за цикъл.

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 ← 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 $n+1$

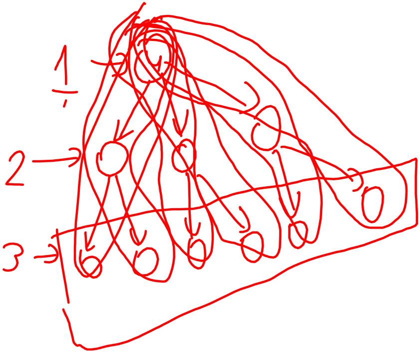


Схема на обхождане в ширина

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

Схема на обхождане в ширина

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

Схема на обхождане в ширина

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

- Какво се случва ако графът е цикличен?

Схема на обхождане в ширина

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

- Какво се случва ако графът е цикличен?
  - Ако има път: намира го.

Схема на обхождане в ширина

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

Схема на обхождане в ширина

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

Схема на обхождане в ширина

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

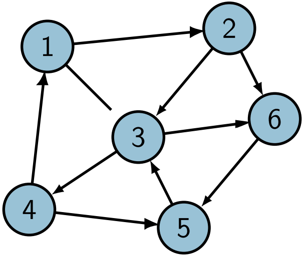
- последно посетеният възел е най-лесно достъпен

Разширяване на пътища

Удобно е пътищата да са представени като стек

- последно посетеният възел е най-лесно достъпен

`(extend '(2 1))` $\longrightarrow$ `((3 2 1) (6 2 1))`



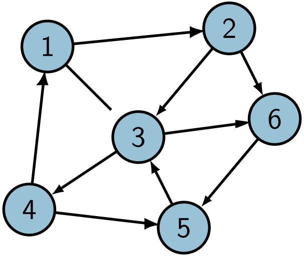
Разширяване на пътища

Удобно е пътищата да са представени като стек

- последно посетеният 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))
```



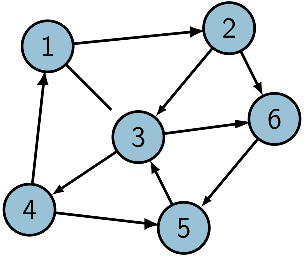
Разширяване на пътища

Удобно е пътищата да са представени като стек

- последно посетеният 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))
```



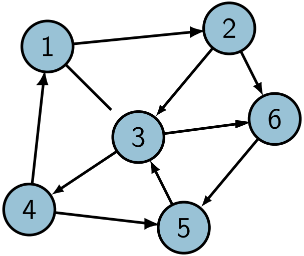
Трябва да филтрираме циклите:

Разширяване на пътища

Удобно е пътищата да са представени като стек

- последно посетеният 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.

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.

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)
    (or (search target-path level)
        (bfs-level (extend-level level))))

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