Лекция 4
Структуры данных

Потребность в структурах данных

- Рассмотрим пример: арифметика рациональных чисел
- Реализация без структуры данных

```
(define (num-of-sum num1 den1 num2 den2)
  (+ (* num1 den2) (* num2 den1)))
(define (den-of-sum num1 den1 num2 den2)
  (* den1 den2))
```

- недостатки:
  - отслеживание какие числители соответствуют каким знаменателям
  - большое количество аргументов функций
  - все операции из двух частей – вычисление числителя и вычисление знаменателя

Проектирование структуры данных

- Придумаем способ работы с рациональными числами.
- Основные операции (конструктор и селекторы):
  - (make-rat num den) – создать рац. число
  - (num r) – получить числитель числа r
  - (den r) – получить знаменатель числа r
- Полагая операции реализованными можно определить сумму, произведение …

```
(define (sum-rat a b)
    (make-rat (+ (* (num a) (den b)) (* (den a) (num b)))
              (* (den a) (den b))))
(define (mul-rat a b)
    (make-rat (* (num a) (num b)) (* (den a) (den b))))
```

Проектирование структуры данных

- sum-rat такова, что 1/2 + 1/6 = 8/12, а не 2/3
- Можно переписать sum-rat, используя gcd

```
(define (sum-rat a b)
   (let* ((n (+ (* (num a) (den b)) (* (den a) (num b))))
                (d (* (den a) (den b)))
                (g (gcd n d)))
          (make-rat (/ n g) (/ d g))))
```

- Можно специальным образом определить make-rat (или селекторы):

```
(define (make-rat a b)
     (let ((g (gcd a b))) (cons (/ a g) (/ b g))))
(define num car)
(define den cdr)
```

Что дала структура данных?

- Получилась слоистая система:

программы, работающие
с рациональными числами

---

sum-rat, mul-rat,..

---

make-rat, num, den

---

cons, car, cdr

реализация пар

Что дала слоистая система?

- Изменения локализованы на уровне
- Переопределим make-rat и селекторы:

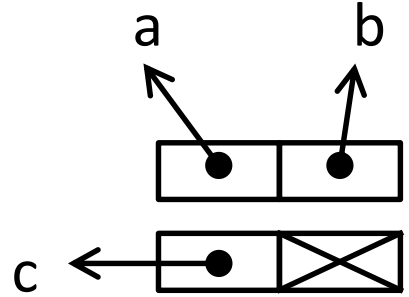
```
(define make-rat cons)
(define (num x)
    (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (den x)
    (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

- sum-rat, mul-rat работают, их не надо переписывать

cons – кирпичик структур данных

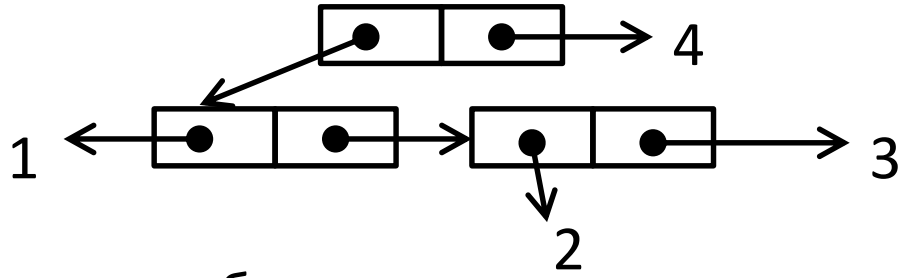
- cons создаёт точечную пару
- внешнее представление > (cons 'a 'b) -> (a . b)
- пара – не всегда список!

> (cons 'c '()) -> (c)



- элементами пары может быть всё, что угодно, в том числе другие пары:

> (cons (cons 1 (cons 2 3)) 4) ->
((1 2 . 3) . 4)



- свойство замыкания: результаты работы cons можно сconsить.

Отступление. Cons, car, cdr и функции

- Рассмотрим код

```
(define (cons x y)
    (define (dispatch m)
      (cond ((= m 0) x)
            ((= m 1) y)
            (else (error "ошибка cons"))))
    dispatch)
(define (car x) (x 0))
(define (cdr x) (x 1))
```

- Получили «функциональную» реализацию пар.
- Вообще говоря, можно числа реализовать функциями…
- …а значит – и все данные.

Отступление. Cons, car, cdr и функции

- Другая «функциональная» реализация пар

```
(define (cons x y)
          (lambda (m) (m x y)))
(define (car pair) (pair (lambda (p q) p)))
(define (cdr pair) (pair (lambda (p q) q)))

> (car (cons 1 2))
--> (car (lambda (m) (m 1 2)))
--> ((lambda (m) (m 1 2)) (lambda (p q) p))
--> ((lambda (p q) p) 1 2)
-> 1
```

аналогично cdr

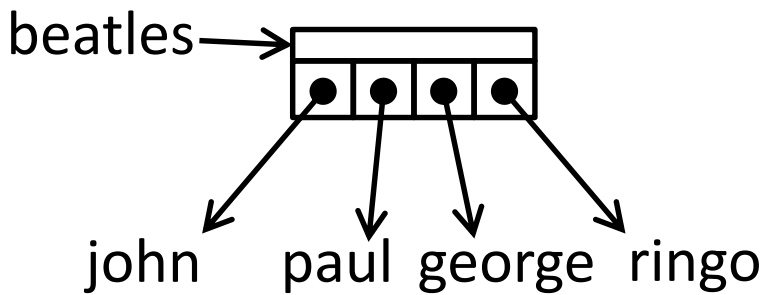
Векторы (пока немутируемые)

- Вектор – массив (фиксированного размера коллекция значений с доступом по индексу)
- конструктор (vector-immutable <e_0> ... <e_n>) или просто vector
- селектор (vector-ref <vector> <index>)
- длина (vector-length <vector>)
- внешнее представление #(val_0 val_1 val_2 … val_n)
- проверка на вектор (vector? vect)

```
> (define beatles (vector-immutable 'john 'paul 'george 'ringo))
> beatles -> #(john paul george ringo)
> (vector-length beatles) -> 4
> (vector-ref beatles 1) -> paul
```

- пустой вектор #()

```
> (vector) -> #()
```



Векторы

- интерпретатор съедает вектора во внешнем представлении, при этом # сама работает как апостроф, пример:

> (define x #(+ - * /)) ;вектор имён арифметических функций

> (vector-ref x 1) -> - ;элемент – имя как символ

> ((eval (vector-ref x 1)) 10) -> -10 ;вычисляем eval-ом, применяем

- преобразование (list->vector <list>)
- преобразование (vector->list <vector>)
- дополнительные функции: vector-map, vector-append, vector-filter, vector-count, vector-argmin, vector-argmax, vector-member, build-vector

полезна директива (require racket/vector)

Деревья

- Будем рассматривать бинарные деревья с данными как в узлах, так и в листьях.
- Дерево, это такая абстракция, для которой верно:
  - Пустое дерево – это дерево. empty-tree
  - Непустое дерево состоит из корня и двух поддеревьев.

  (make-tree data left right)
  - Дерево можно проверить на пустоту. empty-tree?
  - Селекторы дерева:
    - получить корень tree-data
    - получить левое поддерево tree-left
    - получить правое поддерево tree-right

Деревья

- Полагая базовые операции реализованными, можно определять более сложные:
  - проверить, все ли узлы удовлетворяют определенному условию:

```
(define (all-tree p t)
  (or (empty-tree? t)
      (and (p (tree-data t)) (all-tree p (tree-left t))
                             (all-tree p (tree-right t)))))
```

  - существует ли узел, удовлетворяющий условию:

```
(define (any-tree p t)
  (not (all-tree (lambda (x) (not (p x))) t)))
```

стоит ли так писать? ☺

Деревья

- Получить список всех узлов:

```
(define (flatten-tree t)
  (if (empty-tree? t) '()
      (append (flatten-tree (tree-left t))
              (cons (tree-data t) (flatten-tree (tree-right t))))))
```

- Получить список листьев:

```
(define (fringe-tree t)
  (cond ((empty-tree? t) '())
      ((and (empty-tree? (tree-left t))
            (empty-tree? (tree-right t)))
       (list (tree-data t)))
     (else (append (fringe-tree (tree-left t))
                   (fringe-tree (tree-right t))))))
```

Деревья

- Получить список узлов, удовлетворяющих условию:

```
(define (collect-tree p t)
  (cond ((empty-tree? t) '())
        ((p (tree-data t))
         (append (collect-tree p (tree-left t))
                 (cons (tree-data t) (collect-tree p (tree-right t)))))
        (else
         (append (collect-tree p (tree-left t))
                 (collect-tree p (tree-right t))))))
```

Деревья

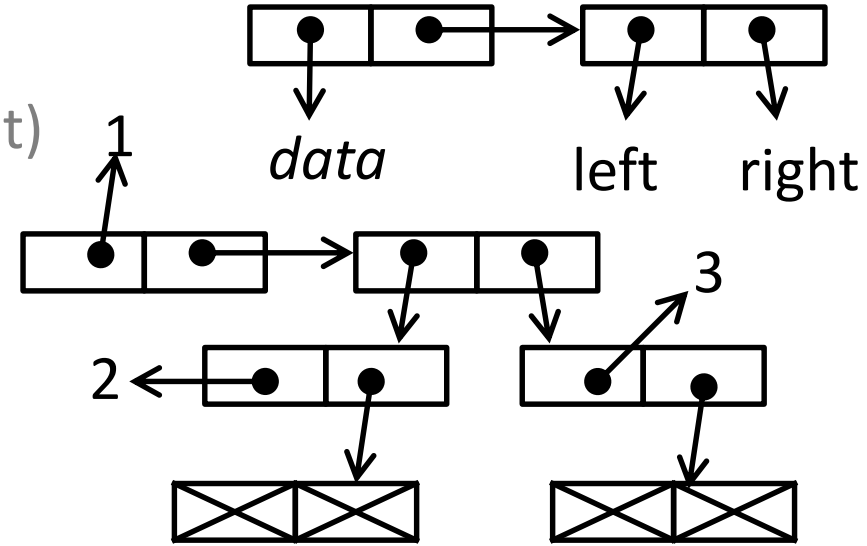
- Выполнить действие для каждого узла, удовлетворяющего условию:

```
(define (with-all-satisfying p k t)
  (let helper ((t t))
    (if (empty-tree? t)
        #f
        (begin
          (if (p (tree-data t)) (k (tree-data t)) #f)
          (helper (tree-left t))
          (helper (tree-right t))))))
 )
```

Деревья

- Реализуем базовые операции:

```
(define empty-tree '())
(define (make-tree data left right)
  (cons data (cons left right)))
(define tree-data  car)
(define tree-left  cadr)
(define tree-right cddr)
(define empty-tree? null?)
```



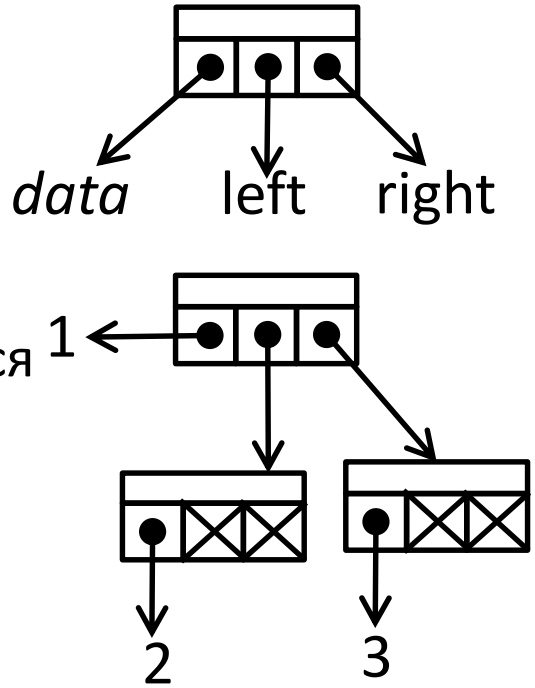
- Особенность внешнего представления

(make-tree 1 (make-tree 2 empty-tree empty-tree) (make-tree 3 empty-tree empty-tree)) -> (1 (2 ()) 3 ())

Деревья

- В conse лишь 2 «ящика», а в векторе -- столько, сколько надо.
- Реализуем базовые операции в «векторном» представлении

```
(define empty-tree #())
(define make-tree vector)
(define (tree-data tree) (vector-ref tree 0))
(define (tree-left tree) (vector-ref tree 1))
(define (tree-right tree) (vector-ref tree 2))
(define (empty-tree? t) (equal? t #())) ; eq? не годится
```



- Теперь вывод лучше:

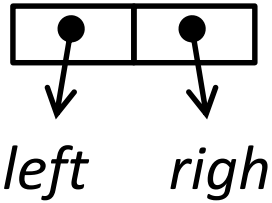
```
> (make-tree 1 (make-tree 2 empty-tree empty-tree)
(make-tree 3 empty-tree empty-tree))
-> #(1 #(2 #() #()) #(3 #() #()))
```

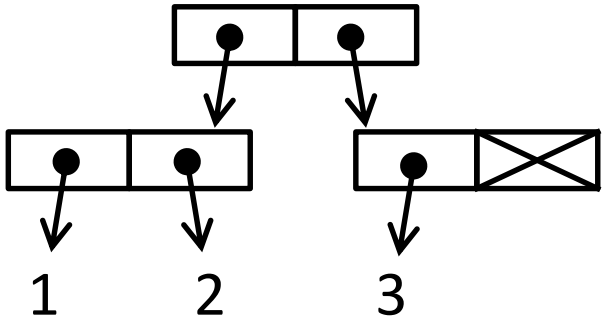
- И списки, и вектора подходят для внутреннего представления.

Деревья

- Иногда рассматривают деревья с данными только в листьях:

```
(define empty-tree '())
(define make-tree cons)
(define tree-left  car)
(define tree-right cdr)
(define tree-empty? null?)
(define (leaf-tree? x) (not (pair? x)))
(define (tree-data t)
          (if (leaf-tree? t) t '()))
```





Пример построения дерева:

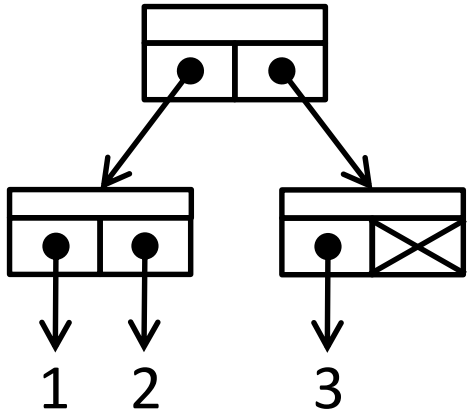
```
> (make-tree (make-tree 1 2) (make-tree 3 empty-tree))
-> ((1 . 2) 3)
```

Снова стандартный вывод невполне удобен.

Деревья

- Векторы-деревья с данными только в листьях:

```
(define empty-tree #())
(define make-tree vector)
(define (tree-left tree) (vector-ref tree 0))
(define (tree-right tree) (vector-ref tree 1))
(define (empty-tree? tree) (equal? tree #())
(define (leaf-tree? tree) (not (vector? tree)))
(define (tree-data tree)
          (if (leaf-tree? tree) tree '()))
```



- Реализация векторами даёт только prettyprinting.

```
> (make-tree (make-tree 1 2) (make-tree 3 empty-tree))
-> #(#(1 2) #(3 #()))
```

Бинарные деревья поиска

- В дереве поиска данные/ключи упорядочены (меньшие – слева от корня, большие – справа).
- Базовые операции дерева поиска:
  - empty-bst – пустое дерево
  - (empty-bst? tree)
  - (insert-bst key tree)
  - (member-bst? key tree)
  - (delete-bst? key tree) ; без реализации

Бинарные деревья поиска

- Реализации

```
(define empty-bst empty-tree)
(define empty-bst? empty-tree?)
(define (member-bst? key t)
          (cond ((empty-bst? t) #f)
      ((= key (tree-data t)) #t)
      ((< key (tree-data t))
       (member-bst? key (tree-left t)))
      (else
       (member-bst? key (tree-right t)))))
```

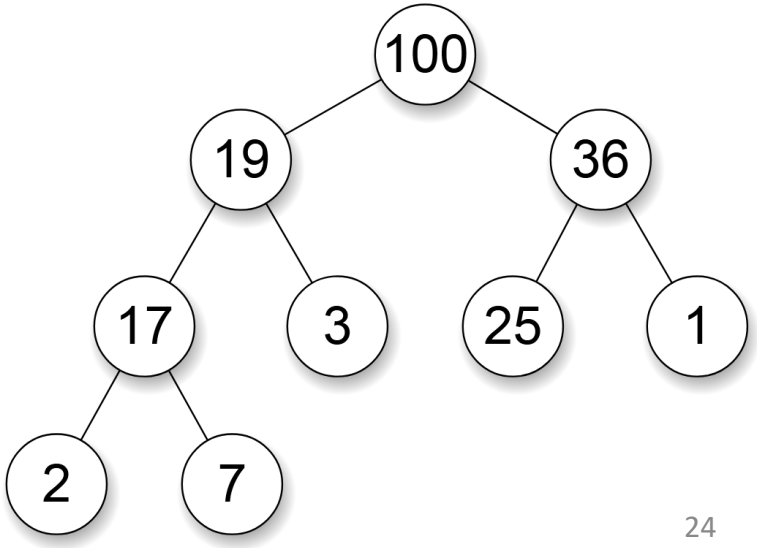
Бинарные деревья поиска

- Вставка (без балансировки)

```
(define (insert-bst key t)
  (cond ((empty-bst? t)
         (make-tree key empty-bst empty-bst))
        ((= key (tree-data t)) t)
        ((< key (tree-data t)) (make-tree (tree-data t)
                                          (insert-bst key (tree-left t))
                                          (tree-right t)))
        (else (make-tree (tree-data t)
                         (tree-left t)
                         (insert-bst key (tree-right t))))))
```

Бинарные деревья-кучи

- В дереве-куче данные/ключи упорядочены (потомки не больше, чем родитель).
- Дерево-куча сбалансировано (поддеревья при любой нелистовой вершине различаются по высоте не больше, чем на 1).
- Базовые операции дерева-кучи:
  - empty-heap – пустая куча
  - (empty-heap? tree)
  - (find-max tree)
  - (insert-heap key tree)
  - (delete-max tree)
- Кучи полезны в некоторых задачах.
  Пример: пирамидальная сортировка



Бинарные деревья-кучи

- Реализации empty-heap, empty-heap?, find-max, insert-heap

```
(define empty-heap empty-tree)
(define empty-heap? empty-tree?)
(define find-max tree-data)
(define (insert-heap key tree)
  (if (empty-heap? tree) (make-tree key empty-heap empty-heap)
      (let ((h (find-max tree)))
        (if (> key h) (make-tree key (insert-heap h (tree-right tree))
                                     (tree-left tree))
                      (make-tree h (insert-heap key (tree-right tree))
                                   (tree-left tree))
      ))))
```

Бинарные деревья-кучи

- Удаление (неэффективная реализация, но существует O(log(n)))

```
(define (delete-max tree)
  (cond ((empty-tree? tree) tree)
        ((empty-tree? (tree-left tree)) (tree-right tree))
        ((empty-tree? (tree-right tree)) (tree-left tree))
        (else (let ((left (find-max (tree-left tree)))
                    (right (find-max (tree-right tree))))
                (if (>= left right) (make-tree left (tree-right tree)
                                               (delete-max (tree-left tree)))
                                    (make-tree right
                                               (insert-heap left (delete-max (tree-right tree)))
                                               (delete-max (tree-left tree)))
)))))
```

Готовые реализации бинарных деревьев-куч

```
(require data/leftist-tree)
(leftist-tree <=? lst) ; <=? – функция сравнения, lst -- список элементов
(leftist-tree-empty? tree) ; чеккер на пустоту
(leftist-tree-add tree a) ; вставка элемента a
(leftist-tree-min tree) ; доступ к корню (без удаления)
(leftist-tree-remove-min tree) ; удаление корня (без доступа)
```

другой вариант:

```
(require pfds/heap/leftist)
(heap <=? e1 ...) ; <=? – функция сравнения, e1 … -- элементы
(empty? heap) ; чеккер на пустоту
(insert a heap) ; вставка
(find-min/max heap) ; доступ к корню (без удаления)
(delete-min/max heap) ; удаление корня (без доступа)
(merge heap1 heap2) ; слияние воедино
```

Итоги лекции 4

- Структуры данных упрощают программы.
- При проектировании структур данных полезно использовать слоистые системы.
- Основа иерархических структур данных в Scheme – точечная пара (cons, car, cdr).
- Вместо точечной пары также может сгодиться *вектор*.
- Многие алгоритмы на деревьях порождают рекурсивные процессы. И это нормально. *Не всегда стоит* пытаться писать итеративные решения, программируя такие алгоритмы.
- *Куча* – полезная структура данных.