Разбор задач анкеты от 21 сентября

- Функция (sqlist n) для натурального n возвращает список квадратов первых n натуральных чисел (1 4 9 … n^2), а для остальных целых n – пустой список.

```
(define (sqlist-r1 n)
  (if (< n 1) '()
      (append (sqlist-r1 (- n 1)) (list (* n n)))))
```

вспомним, что append – линейный ☻

Разбор задач анкеты от 21 сентября

- Перепишем с cons

```
(define (sqlist-r2 n)
  (define (helper i)
    (if (> i n) '()
        (cons (* i i) (helper (+ i 1)))))
  (helper 1))
```

- теперь итеративный

```
(define (sqlist-i n)
  (define (loop i result)
    (if (< i 1) result
        (loop (- i 1) (cons (* i i) result))))
  (loop n '()))
```

Разбор задач анкеты от 21 сентября

- Функция (powlist n) для натурального n возвращает список первых n+1 степеней 2 (1 2 4 … 2^n), а для остальных целых n – пустой список.

```
(define (powlist-r n)
  (define (helper i pow-i)
    (if (> i n) '()
        (cons pow-i (helper (+ i 1) (* pow-i 2)))))
  (if (< n 1) '() (helper 0 1)))
```

Разбор задач анкеты от 21 сентября

- теперь итеративный

```
(define (powlist-i n)
  (define (loop i pow-i result)
    (if (< i 0) result
        (loop (- i 1) (* pow-i 2) (cons pow-i result))))
  (if (< n 1) '() (reverse (loop n 1 '()))))
```

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

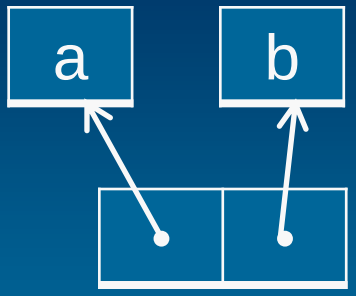
Что дала слоистая система?

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

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

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

cons

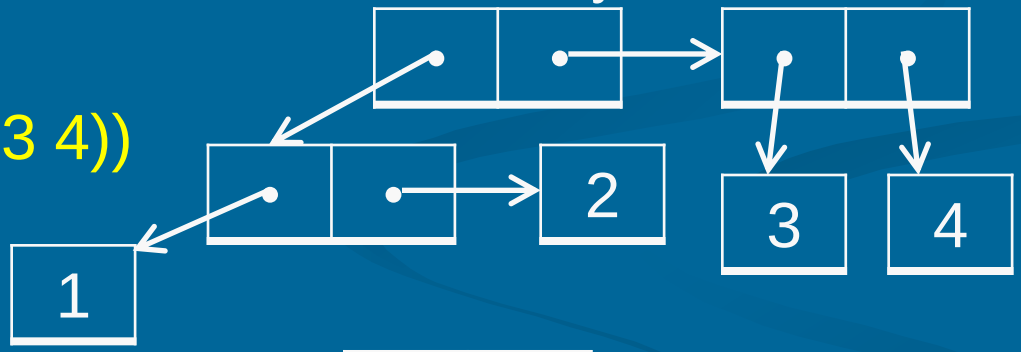


- cons создаёт точечную пару
- внешнее представление (cons a b) => (a . b)
- пара – не всегда список!

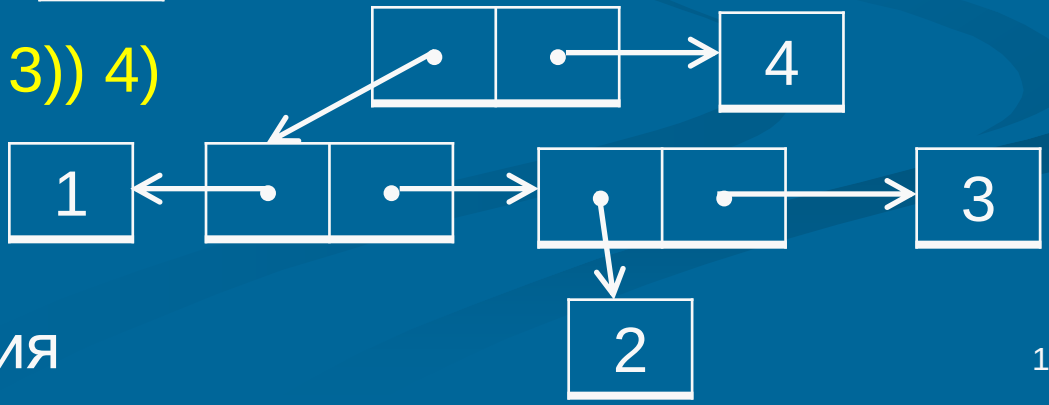
(cons a '())

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

(cons (cons 1 2) (cons 3 4))



(cons (cons 1 (cons 2 3)) 4)



- свойство замыкания

Отступление. Точечные пары и функции

- С помощью нотации точечной пары можно определить функцию с нефиксированным количеством параметров:

```
(define (f . params)
   (display params))
(f) =>                  ()
(f 1 2 3 4 5) =>        (1 2 3 4 5)

(define (first+ x . tail)
   (map (lambda (y) (+ x y)) tail))

(first+ 10 1 2 3 4) =>        (11 12 13 14)
(first+ 10)         =>        ()
```

Отступление. Cons, car, cdr и функции

- Рассмотрим код

```
(define (cons x y)
   (define (dispatch m)
     (cond ((= m 0) x)
           ((= m 1) y)
           (else (println “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

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)
  (if (empty-tree? t) #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))
```

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)
  (define (rec t)
    (if (empty-tree? t)
        #f
        (begin
          (if (p (tree-data t)) (k (tree-data t)) #f)
          (rec (tree-left t))
          (rec (tree-right t)))))
  (rec t))
```

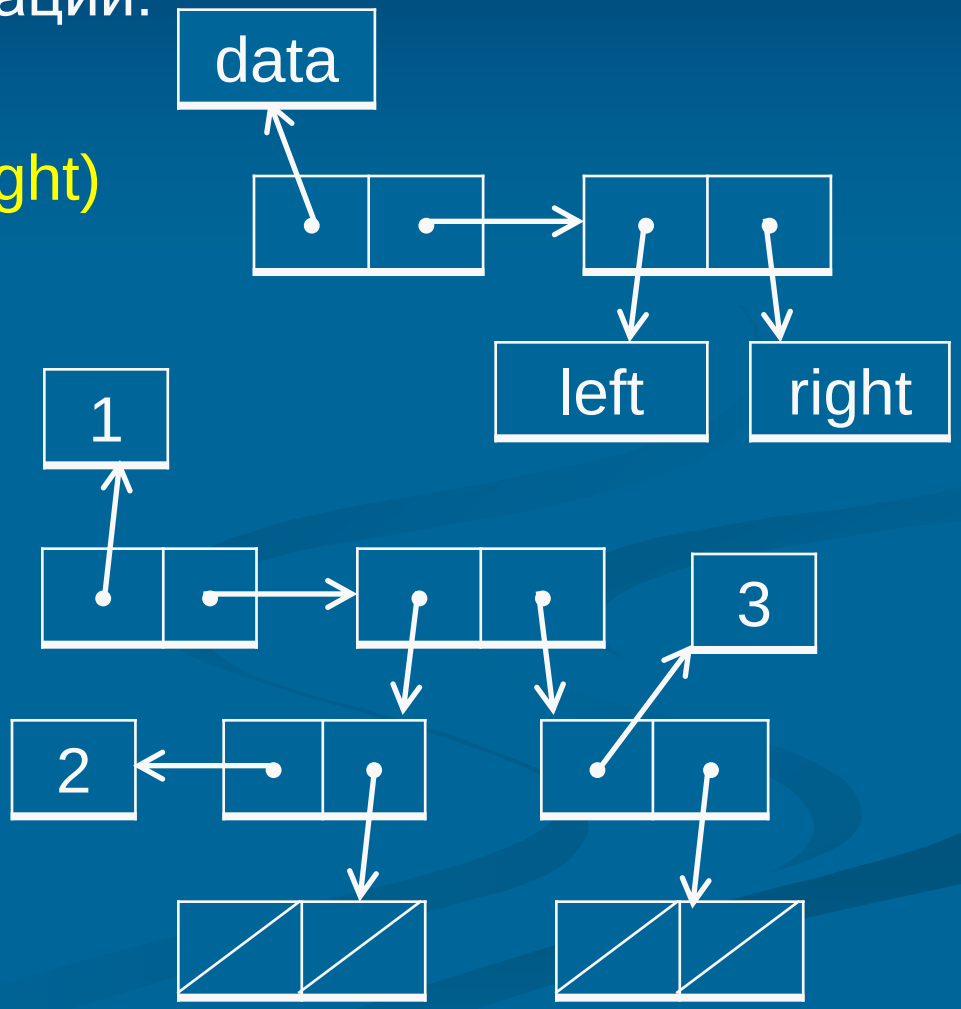
Деревья

- 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? t)
  (eq? t '()))
```

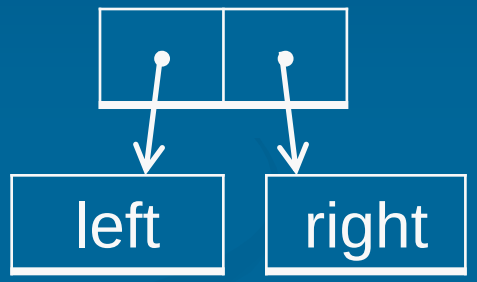


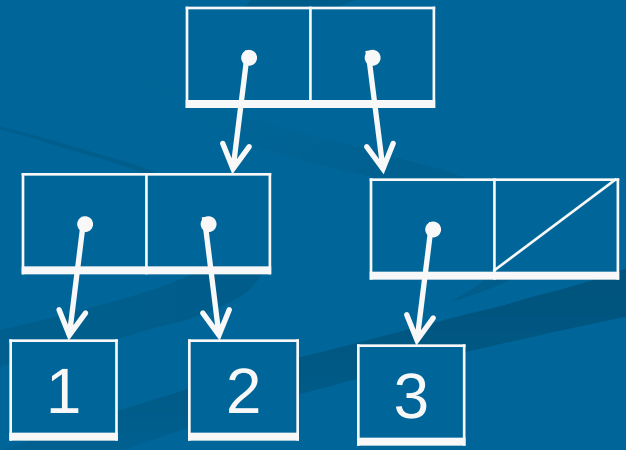
Деревья

- Иногда рассматривают деревья с данными только в листьях:

```
(define empty-tree '())
(define (make-tree left right)
  (cons left right))
(define tree-left  car)
(define tree-right cdr)
(define (empty-tree? t)
  (eq? t '()))

(define (leaf-tree? x) (not (pair? x)))
(define (tree-data t)
          (if (leaf-tree? t) t '()))
```





Бинарные деревья поиска

- В дереве поиска данные упорядочены (меньшие – слева от корня, большие -- справа).
- Базовые операции дерева поиска:
  - empty-bst
  - (empty-bst? t)
  - (insert-bst key t)
  - (member-bst? key t)

Бинарные деревья поиска

- Реализации

```
(define empty-bst empty-tree)

(define (empty-bst? t) (empty-tree? t))

(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))))))
```

Итоги лекции 5

- Структуры данных упрощают программы
- При проектировании структур данных полезно использовать слоистые системы.
- Основа иерархических структур данных в Scheme – точечная пара (cons, car, cdr).