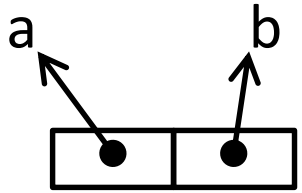
Отступление: Cons и стрелочные диаграммы

- cons создаёт точечную пару
- внешнее представление



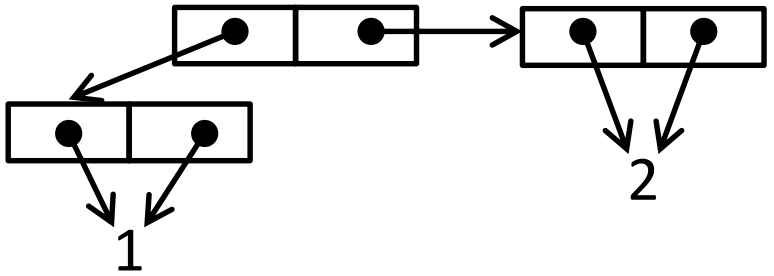
> (cons 'a 'b) -> (a . b)

- пара – не всегда список!

список – пара со 2-м элементом-списком (cons 'c '())

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

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

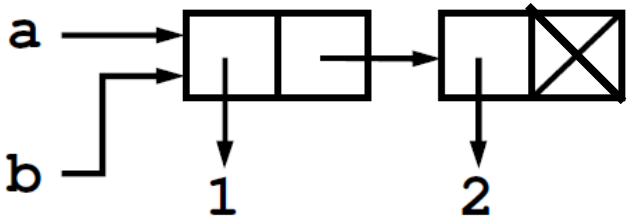


- у cons есть свойство замыкания (т. е. результаты cons можно сconsить)

Отступление: Тождественность, эквивалентность

- Два объекта могут совпадать

> (eq? a b) -> #t



- Два объекта могут одинаково выглядеть

> (equal? (list 1 2) (list 1 2)) -> #t

> (eq? (list 1 2) (list 1 2)) -> #f

но не совпадать

- Ещё бывает eqv?. Если (eq? x y), то (eqv? x y).

Если (eqv? x y), то (equal? x y).

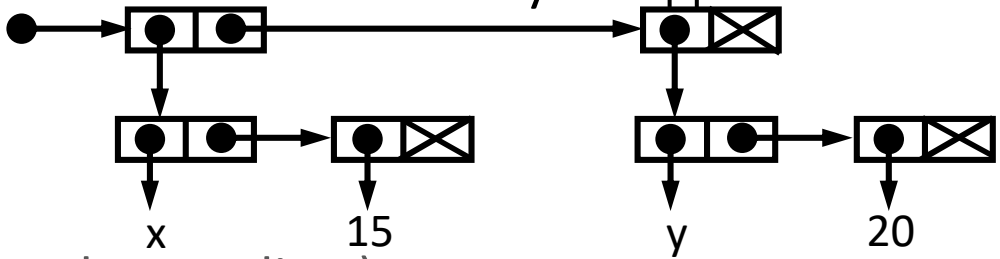
- Но (eq? 1/2 (/ 1 2))≠(eqv? 1/2 (/ 1 2)),

(equal? (cons 1 2) (cons 1 2))≠(eqv? (cons 1 2) (cons 1 2))

Отступление: Assoc. Ассоциативные списки

- списки пар, в которых первый элемент используется для поиска

'((x 15) (y 20))



- поиск пары по equal? (assoc <key> <alist>)

> (assoc 'x '((w 1) (x 2) (y 3) (z 4))) -> (x 2)

- искать по eq? (assq <key> <alist>)

> (assq 'x '((w 1) (x 2) (y 3) (z 4))) -> (x 2)

> (assq '(x 1) '(((x 1) 2) ((y 1) 3)))) -> #f

> (assoc '(x 1) '(((x 1) 2) ((y 1) 3)))) -> ((x 1) 2)

- искать по своему сравнению (assoc <key> <alist> <fun>)
- искать по предикату (assf <pred> <alist>)

ЛЕКЦИЯ 3
Функции высшего порядка

Функция высшего порядка

- Функция, манипулирующая другими функциями, называется функцией высшего порядка.
- Пример -- суммирование:

```
(define (sum-squares a b) ; функция 1го порядка
        (if (> a b)
                0
                (+ (* a a) (sum-squares (+ a 1) b))))
(define (sum-cubes a b) ; функция 1го порядка
        (if (> a b)
                0
                (+ (* a a a) (sum-cubes (+ a 1) b))))
```

Функция высшего порядка

- Суммирование (продолжение) $\sum 1/((4i-3)*(4i-1))$:

```
(define (pi-sum a b) ; функция 1го порядка
        (if (> a b)
                0
                (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

- Проглядывается общая схема

```
(define (<имя> a b)
        (if (> a b)
                0
                (+ (<терм> a) (<имя> (<следующий> a) b))))
```

Функция высшего порядка

- Опишем функцию суммирования:

```
(define (sum term a next b) ; функция высшего порядка
        (if (> a b) 0
                (+ (term a) (sum term (next a) next b))))
```

- Сумма кубов через sum:

```
(define (sum-cubes a b)
        (sum (lambda (x) (* x x x)) a (lambda (x) (+ x 1)) b))
```

- Сумма квадратов через sum

```
(define (sum-squares a b)
        (sum (lambda (x) (* x x)) a add1 b)) ; add1 делает инкремент
```

- (define (pi-sum a b)

```
(sum (lambda (x) (/ 1.0 (* x (+ x 2)))) a (lambda (x) (+ x 4)) b))
```

«лайфхак»: можно использовать анонимные функции!

Итог по суммированию

- Описали схему суммирования функцией высшего порядка.
- Повысили уровень абстракции.
- Избавились от дублирования кода.
- Явно выразили идею в программе.

- sum порождает рекурсивный процесс, перепишем:

```
(define (sum term a next b)
    (let loop ((a a) (result 0))
          (if (> a b) result
                (loop (next a)
                      (+ (term a) result))))
)
```

Перемножение

- По аналогии с суммированием можно умножать:

```
(define (product term a next b)
    (let loop ((a a) (result 1))
        (if (> a b)
            result
            (loop (next a) (* (term a) result))))
)
(define (sum term a next b)        ; есть что-то общее / совпадение???
    (let loop ((a a) (result 0))
        (if (> a b)
            result
            (loop (next a) (+ (term a) result))))
)
```

Накопление

- Повысим уровень абстракции:

```
(define (accumulate combiner null-val term a next b)
    (let loop ((a a) (result null-val))
            (if (> a b)
                    result
                    (loop (next a) (combiner (term a) result))))
)

(define (sum term a next b)
    (accumulate + 0 term a next b))

(define (product term a next b)
    (accumulate * 1 term a next b))
```

Накопление

- Пример использования *accumulate*:

```
(define (enumerate-interval a b)
    (accumulate (lambda (x y) (append y x))
            '() list a add1 b))
```

из-за append enumerate-interval работает за квадрат!

```
> (enumerate-interval 1 10)       -> (1 2 3 4 5 6 7 8 9 10)
```

- Если в обратном порядке, то проще

```
> (accumulate cons '() (lambda (x) x) 10 add1 20)
->  (20 19 18 17 16 15 14 13 12 11 10)
```

cons лучше append’а, т. к. даёт линейное решение!

- если всё ещё нужно в прямом, то считаем в обратном и переворачиваем через reverse, который линейный.

Накопление с фильтром

- Опишем filtered-accumulate:

(define (filtered-accumulate *predicate* combiner null-val term a next b) …)

если очередное *a* удовлетворяет фильтру, происходит накопление, иначе пропускаем и берем следующее *a*.

```
(define (filtered-accumulate predicate combiner null-val term a next b)
    (let loop ((a a) (result null-val))
          (if (> a b)          result
                               (loop (next a)
                               (if (predicate a) (combiner (term a) result)
                                                 result))))
)
```

Функция возвращает функцию

- Определим взятие производной от функции-аргумента:

```
(define (derive f)
        (lambda (x) (/ (- (f (+ x 0.00001)) (f x)) 0.00001)))
```

- Вызовем: ((derive (lambda (x) (* x x))) 10)
- По подстановочной модели:

```
-->((lambda (x) (/ (- ((lambda (x) (* x x)) (+ x 0.00001)) ((lambda (x) (* x
    x)) x)) 0.00001)) 10)
-->
(/ (- ((lambda (x) (* x x)) (+ 10 0.00001)) ((lambda (x) (* x x)) 10))
    0.00001))
...
-> 20.00000999942131  ~ 2x в точке x = 10
```

Функции высшего порядка и списки

- Умножение элементов списка

```
(define (scale-list lst factor)
    (if (null? lst) '()
            (cons (* (car lst) factor) (scale-list (cdr lst) factor))))
```

- Приращение, возведение в степень …
- Отображение списка *(map <функция> <список>)*

```
(define (map func lst)
    (let loop ((lst lst) (result null))
        (if (null? lst) (reverse result)
            (loop (cdr lst) (cons (func (car lst)) result)))))
(define (scale-list lst factor)
    (map (lambda (x) (* x factor)) lst))
```

Функции высшего порядка и списки

- Стандартный *map*

(map <функция> <список1> <список2> … <списокN>)

```
> (map + '(1 2 3) '(10 20 30) '(100 200 300))
->  (111 222 333)
```

- «Близнец» map – *for-each*

```
> (for-each (lambda (x) (println x)) '(1 2 3))
->          1
            2
            3
```

for-each не возвращает список!

Функции высшего порядка и списки

- Просеивание filter

(filter <предикат> <список>)

```
> (filter odd? '(1 2 3 4 5))
->  (1 3 5)
(define (filter predicate lst)
  (let loop ((lst lst) (result null))
      (cond ((null? lst) (reverse result))
                ((predicate (car lst))
                          (loop (cdr lst) (cons (car lst) result)))
                (else (loop (cdr lst) result)))))
```

Функции высшего порядка и списки

- Накопление

(accum <функция> <нач значение> <список>)

```
> (accum + 0 '(1 2 3 4 5))
->  15
> (accum * 1 '(1 2 3 4 5))
->  120
(define (accum func init lst)
    (if (null? lst) init
            (func (car lst) (accum func init (cdr lst)))))
```

Стандартный синоним foldr. Accum – учебное имя. Его в программах не используем!

Правоассоциативная свертка

(func e1 (func e2 (... (func eN init)…) ; рекурсивный!!!

Функции высшего порядка и списки

- Левоассоциативная свёртка

(foldl <функция> <нач значение> <список>)

```
(define (foldl func init lst)
    (if (null? lst) init
            (foldl func (func (car lst) init) (cdr lst))))
```

(func eN (func eN-1 (... (func e1 init)...))) ; итерационный!!!

Если можно, то вместо foldr используем foldl.

> (foldl cons '() '(1 2 3 4)) -> (4 3 2 1)

Но

> (foldr cons '() '(1 2 3 4)) -> (1 2 3 4)

> (foldl list '() '(1 2 3 4)) -> (4 (3 (2 (1 ()))))

> (foldr list '() '(1 2 3 4)) -> (1 (2 (3 (4 ()))))

Лево- и право- ассоциативные свёртки

- сумма списка

(foldl + 0 lst) ; найдём сумму подсписка без последнего элемента и прибавим последний элемент

(foldr + 0 lst) ; найдём сумму хвоста списка и прибавим первый элемент

- минимум списка чисел

(foldl min +inf.0 lst) ; найдём min в подсписке без последнего элемента и выберем минимум из него и последнего элемента

(foldr min +inf.0 lst) ; найдём min в хвосте списка и выберем минимум из него и первого элемента

Когда порядок неважен, «левая» свёртка лучше «правой»!

Лево- и право- ассоциативные свёртки

- (map func lst) реализация через свёртку

(foldl (lambda (x y) (append y (list (func x)))) '() lst)

; найдём map подсписка без последнего элемента и допишем к нему список из func от последнего элемента

Чтобы не было append, лучше получить результат в обратном порядке, а затем перевернуть

(reverse (foldl (lambda (x y) (cons (func x) y)) '() lst))

Вообще говоря, reverse тоже реализуется foldl.

- (map func lst) реализация через другую свёртку

(foldr (lambda (x y) (cons (func x) y)) '() lst)

; найдём map от хвоста списка припишем его справа после func от первого элемента

Функции высшего порядка и списки

- map, filter и свертки позволяют легко реализовывать обработку списков

- произведение квадратов нечетных чисел из списка

```
(define (product-of-squares-of-odd-elems lst)
    (foldl * 1
           (map (lambda (x) (* x x))
                (filter odd? lst))))
; но нужны ли тут 3 прохода?
    (foldl (lambda (x y) (if (odd? x) (* y x x) y)) 1 lst)

> (product-of-squares-of-odd-elems '(1 2 3 4 5)) -> 225
```

Функции высшего порядка и списки

- список квадратов первых n чисел Фибоначчи

```
(define (list-fib-squares n)
        (map (lambda (x)
                  (let ((temp  (fib x))) (* temp temp)))
                     (enumerate-interval 1 n)))
; неэффективно!
```

Д/з

1) Написать линейно-итерационно.
2) Написать линейно-итерационно со свёрткой. ☺

Функции высшего порядка и списки

- Другие функции высшего порядка для списков
- *(andmap <функция> <список1> <список2> …)*

вернёт #f, если одно из применений функции даст #f и дальше не будет считать;
если ни одно из применений не #f, то вернёт результат функции на последних элементах; вернёт #t на '()

```
> (andmap positive? '(1 2 a)) -> error
> (andmap positive? '(1 -2 a)) -> #f
```

- *(ormap <функция> <список1> <список2> …)*

вернёт #f, если всюду #f; иначе вернёт первый не #f и дальше не будет считать; вернёт #f на '()

- Ещё бывают после (require racket/list) доп. функции: filter-map, count, append-map, filter-not, argmin, argmax, … .

Пример

- Вычислить значение многочлена в точке: коэффициенты многочлена задаются списком.

```
(define (gorner-l lst x)
; a_n, a_n-1, … a_1, a_0 по убыванию степеней
  (foldl (lambda (a b) (+ (* b x) a)) 0 lst))

(define (gorner-r lst x)
; a_0, a_1, … a_n-1, a_n по возрастанию степеней
  (foldr (lambda (a b) (+ (* b x) a)) 0 lst))
; но лучше с foldl и reverse
; (foldl (lambda (a b) (+ (* b x) a)) 0 (reverse lst)))
```

Снова д/з

- Функция (process lst) получает непустой список списков lst и возвращает список, составленный из следующих по порядку всех списков-элементов lst, сумма элементов которых больше произведения элементов первого списка-элемента lst. Пусть сумма пустого списка равна 0. Пусть произведение пустого списка равно 1. Реализация должна быть эффективной и использующей уместные функции высшего порядка для обработки списков.

Пример:

> (process '((5) (1 2) () (3 4) (2 3) (2 3 4))) -> ((3 4) (2 3 4))

Композиция функций

> (define (my-compose1 f g) (lambda (x) (f (g x))))

> ((my-compose1 add1 -) 10) -> -9

> ((my-compose1 - add1) 10) -> -11

> my-compose1 -> #<procedure>

> (my-compose1 add1 -) -> #<procedure>

- распишем без синтаксического сахара

> (define my-compose1-v2 (lambda (f g) (lambda (x) (f (g x)))))

- добавим нестандартный синтаксический сахар

> (define ((my-compose1-v3 f g) x) (f (g x)))

> ((my-compose1-v3 add1 -) 10) -> -9

(compose1 f1 …) уже есть, мы описали его в учебных целях.

Итоги лекции 3

- Стрелочные диаграммы наглядны.
- Функции высшего порядка – это полезно.
- Абстракция позволяет программисту контролировать сложность программы и явно выражать в тексте свои идеи.
- В Scheme функция может быть как аргументом вызова функции, так и результатом.
- map, filter, foldl, foldr помогают в работе со списками.
- Ещё есть andmap, ormap, filter-map, count, argmin, argmax, append-map, filter-not.
- Вообще говоря, бывают аналоги списковых свёрток для деревьев или подобных динамических структур.