ЛЕКЦИЯ 3

Функции высшего порядка

Функция высшего порядка

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

```
(define (sum-integers a b)
```

```
  (if (> a b)
```

```
      0
```

```
      (+ a (sum-integers (+ a 1) b))))
```

```
(define (sum-cubes a b)
```

```
  (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)
```

```
  (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)
  (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 целых через sum

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

- (define (pi-sum a b)

```
(sum (lambda (x) (/ 1.0 (* x (+ x 2)))) a (lambda (x) (+ x 1)) b))
```

«Лайфхак»: можно использовать λ -выражения!

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

Перемножение

- По аналогии с суммированием можно умножать:

```
(define (product term a next b)
```

```
  (define (loop a result)
```

```
    (if (> a b)
```

```
        result
```

```
        (loop (next a) (* (term a) result))))
```

```
  (loop a 1))
```

```
(define (sum term a next b)
```

; есть что-то общее

```
  (define (loop a result)
```

```
    (if (> a b)
```

```
        result
```

```
        (loop (next a) (+ (term a) result))))
```

```
  (loop a 0))
```

Накопление

- Повысим уровень абстракции:

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

```
(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))
              '() (lambda (x) (list x)) a (lambda (x) (+ x 1)) b))
```

```
(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 (lambda (x) (+ x 1)) 20)
=> (20 19 18 17 16 15 14 13 12 11 10)
```

- если всё ещё нужно в прямом, то считаем в обратном и переворачиваем 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)
```

```
  (define (loop a result)
```

```
    (if (> a b)      result
```

```
        (loop (next a)
```

```
            (if (predicate a) (combiner (term a) result)
                result))))
```

```
  (loop a null-val))
```

Функция высшего порядка 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))
```

...

20.000000999942131 ~ 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)
```

```
  (if (null? lst)
```

```
      '())
```

```
      (cons (func (car lst)) (map func (cdr lst)))))
```

```
(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
```

Функции высшего порядка и списки

- Просеивание filter

```
(filter <предикат> <список>)
```

```
(filter odd? '(1 2 3 4 5))
```

```
=> (1 3 5)
```

```
(define (filter predicate lst)
```

```
  (cond ((null? lst) '())
```

```
        ((predicate (car lst))
```

```
          (cons (car lst) (filter predicate (cdr lst))))
```

```
        (else (filter predicate (cdr lst)))))
```

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`

Правоассоциативная свертка

```
(func e1 (func e2 (... (func eN init)...))
```

Функции высшего порядка и списки

- Левоассоциативная свёртка

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

```
(define (foldl func init lst)
```

```
  (if (null? lst) init
```

```
      (foldl func (func init (car lst)) (cdr lst))))
```

(func eN (func eN-1 (... (func e1 init)...)))

```
(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 неотрицательных чисел

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

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

Лево- и право- ассоциативные свёртки

- (map func lst)

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

найдем map подписка без последнего элемента и допишем к нему список из func от последнего элемента

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

найдем 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 списка

```
(define (product-of-squares-of-odd-elems lst)
```

```
  (foldr * 1
```

```
    (map (lambda (x) (* x x))
```

```
      (filter odd? lst))))
```

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

Функции высшего порядка и списки

- список квадратов чисел Фибоначчи

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

Д/з написать линейно-итерационно.

Пример

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

```
(define (gorner-l lst x)
```

```
;  $a_n, a_{n-1}, \dots, a_1, a_0$  по убыванию степеней
```

```
(foldl (lambda (a b) (+ (* b x) a)) 0 lst))
```

```
(define (gorner-r lst x)
```

```
;  $a_0, a_1, \dots, a_{n-1}, a_n$  по возрастанию степеней
```

```
(foldr (lambda (a b) (+ (* b x) a)) 0 lst))
```

Д/з

- На вход подается список списков, на выходе требуется получить только те списки, сумма элементов которых больше произведения элементов первого списка.

Итоги лекции 3

- 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.
- Вообще говоря, бывают свёртки для деревьев.