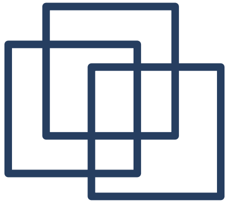
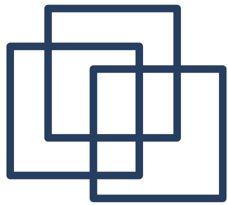


Кафедральный практикум 5 семестр

Часть 3. Функции высшего порядка (continuation-passing style)

http://sp.cmc.msu.ru/~kornykhin/fp/slides/part3-1.pdf



План

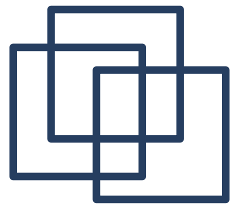
Часть 1. Введение.

Часть 2. Язык программирования Scheme.

Часть 3. Функции высшего порядка. «Векторное» мышление.

1) **continuation-passing style**
2) замыкания (программирование объектами-функциями)
3) свертки над структурами данных
4) программы, управляющие вычислениями
5) . . .

Часть 4. Теоретический фундамент ФП.

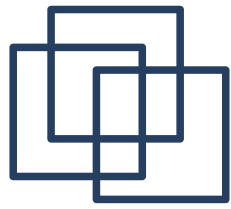


Задачи В.Кулямина

- Распознавание нуля/единицы на картинке
- Валидация XML данных относительно схемы
- Генератор текстов автоматных программ по описанию конечного автомата

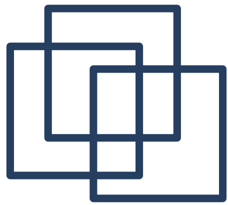
- Успешно справившимся — бонус на экзамене по спецкурсу Виктора Кулямина

- Есть еще спецкурс Дениса Турдакова



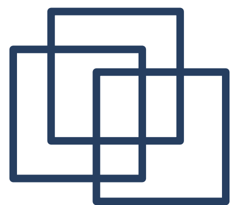
Привычный Scheme

- Программа — набор функций (подпрограмм)
- Что не нравится в программах?
  - долго работают
  - плохочитаемые тексты программ
  - лучше бы писали на C/C++
- Кажется, что Scheme — это такой странный язык, в котором нельзя писать циклы



А на самом деле...

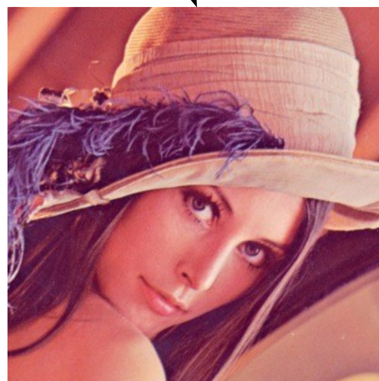
- Scheme всего с добавлением **одной** (!) вещи позволяет делать:
  - нелокальные выходы из процедур
  - эффективная рекурсия (преждевременный выход)
  - перебор с возвратом
  - exception'ы
  - отложенные вычисления
  - callback'и (актуально в Web)
  - сопрограммы
  - многопоточность



Обработать Лену

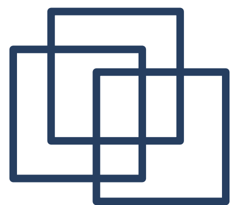
повернуть на 40

обрезать

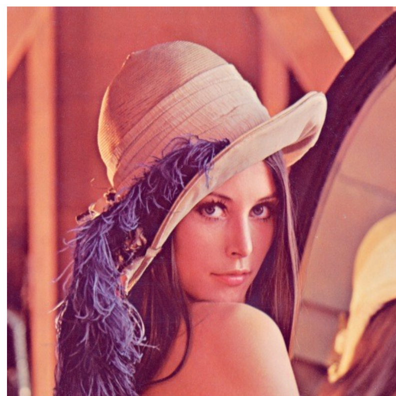


волны

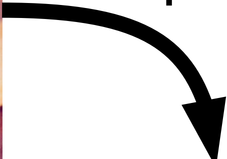
```
(define (обработать Лену)
  (волны
    (обрезать
      (повернуть 40
                 Лену))))
```



Обработать Лену

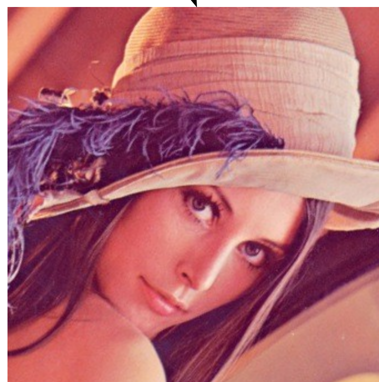


повернуть на 40

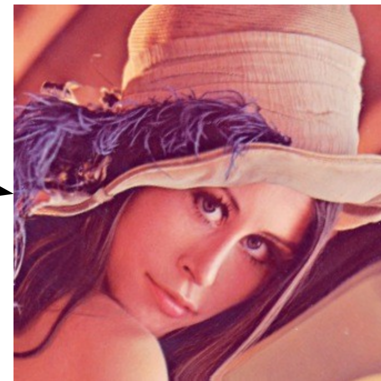


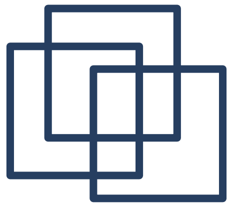
Заметьте!
есть цепочка действий;
за каждым действием
идут «*все остальные действия*»

обрезать



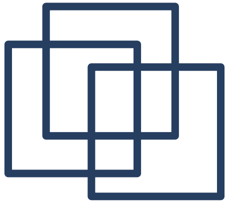
волны





На Scheme

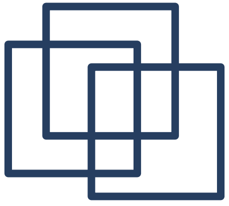
- За каждым действием идут «все остальные действия»:
```
(define (sqr x nexts)
    (nexts (* x x)))
```
- Удобно комбинировать функции (идея ФП в том, чтобы объявлять небольшие функции и их внятно комбинировать):
```
(sqr 1 display)
(sqr 1 print)
(sqr 1 sqr)
```
— хочется записать биквадрат, как это сделать правильно?



Смотрите!

- порядок операций в программе стал человеческим (слева-направо)
  (**sqr** 1 **display**)
  (**sqr** 1 (lambda (s1)
      (**sqr** s1
          display)))

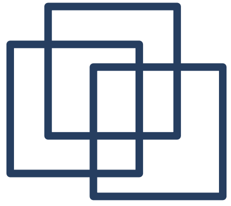
- удобно заменять части алгоритма
  (sqr 1 display)
  (sqr 1 print) — и не надо переписывать sqr, display, print



Упражнение

- записать выражение «прибавить к x y, к ним z, вывести» (читабельность и порядок операций! ничего хитрого!)

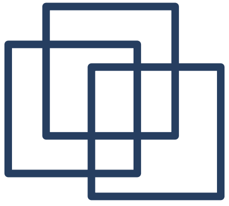
- *Подсказка*: начать со сложения двух чисел: (define (add x y nexts) …)
- NB: nexts — это **все** остальные действия, а не непосредственно следующее за add



Еще упражнение

- записать выражение «прибавить к 1 2, к ним 3, к ним 4, ..., к ним N, вывести»

- есть как минимум 3 принципиально разных рекурсивных решения. Предложите хотя бы два из них

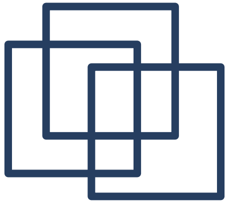


Рекурсия

НАДО ИСПОЛЬЗОВАТЬ

ХВОСТОВУЮ

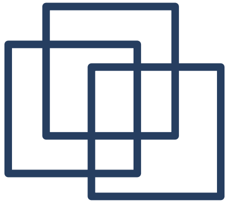
РЕКУРСИЮ



Вставка nexts

- Пусть есть хвосторекурсивная реализация. Можно ли в нее «добавить поддержку nexts»? Как?
- На примере:

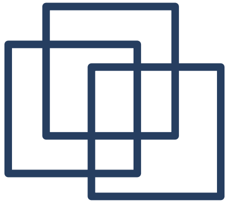
```
(define (reverse xs)
    (define (reverse-tail xs rev)
        (if (null? xs)
            rev
            (reverse-tail (cdr xs) (cons (car xs) rev))))
    (reverse xs '()))
```



CPS

- CPS (continuation-passing style) — это только хвостовая рекурсия с передачей (и наращиванием) «остальных действий» и их вызова в конце рекурсии.

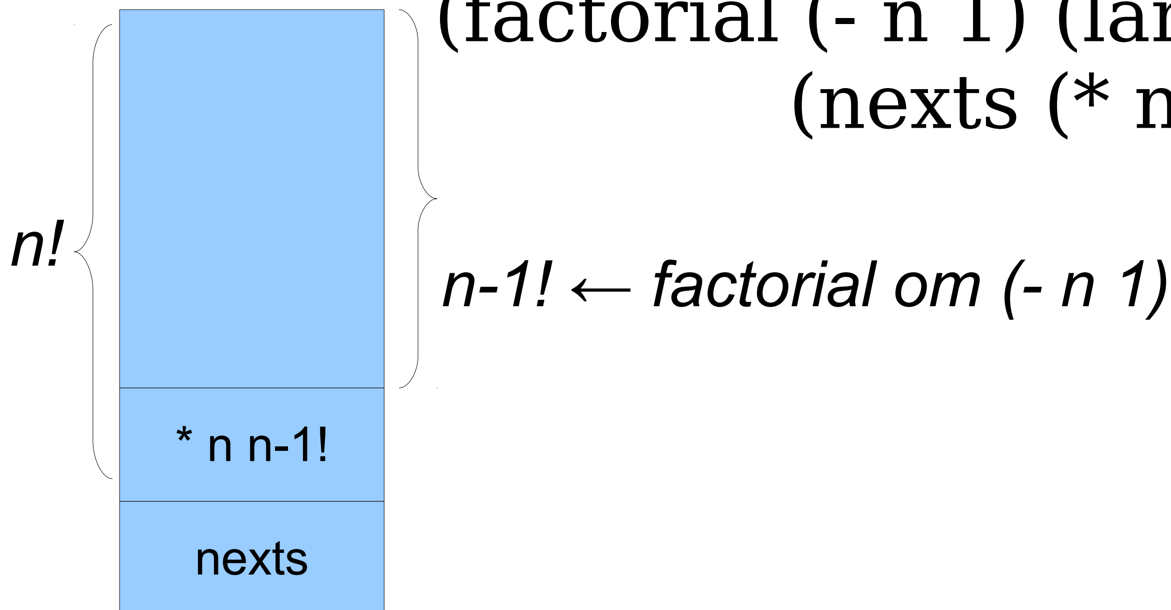
- Зачем?!
  - функция никогда никуда не возвращается! Она *передает управление*.
  - => *можно* обойтись без стека (как? всегда ли?)
  - => дешевый вызов: goto вместо call/return
  - код легче модифицировать (только заменить nexts в вызове)

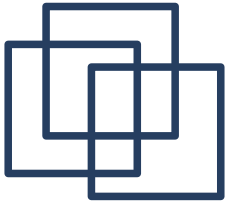


factorial

- Наращивание «остальных действий» :

```
(define (factorial n nexts)
    (if (zero? n) (nexts 0)
          (factorial (- n 1) (lambda (n-1!)
                    (nexts (* n n-1!))))))
```

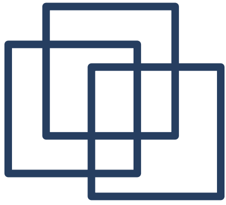




factorial

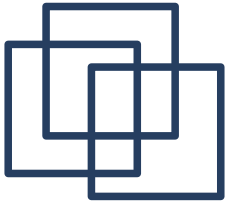
- Решение без «наращивания» :

```
(define (factorial n nexts)
      (define (factorial-tail n prod nexts)
        (if (zero? n) (nexts prod)
          (factorial-tail (- n 1) (* n prod) nexts)))
    (factorial-tail n 1 nexts))
```



Фишки

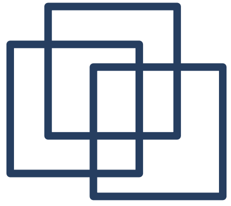
- В CPS легко написать досрочный выход из рекурсии!
- Пример: перемножаем элементы списка. Не обязательно это делать до конца
- ```
  (define (listmult xs nexts)
    (define (listmult-tail xs prod)
      (if (null? xs)        (nexts prod)
      (if (zero? (car xs))(nexts 0)
                          (listmult-tail
                             (cdr xs)
                             (* (car xs) prod)))))
      (listmult-tail xs 1 nexts))
  ```
  ```



Еще раз

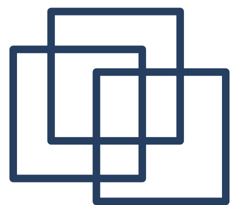
- Определение функций в CPS:
  (define (f x **nexts**) ....)
  в теле функции не вызываем другие функции, а передаем управление

- Использование функций в CPS:
```
(f x (lambda (fx)
    (g y (lambda (gy)
          .... ))
```
  сначала f от x, затем g от y, затем ...



Ложка дёгтя

- На самом деле ничего качественно нового за CPS не стоит, это чисто технический шаг:
  - выявляется порядок вычислений
  - выявляются имена подвыражений
  - оптимизированные вызовы функций
- Для каждой полезной функции нужен ее cps-вариант, впрочем компилятор сам может его построить
- Надо либо сразу писать функции в cps-виде, либо писать в не-cps (и соответственно использовать эти определения)



Домашнее задание

- ejudge : http://earth.ispras.ru
- Задачи требуется решать в continuation-passing style
- В решении запрещается использовать
    - set! и его аналоги
    - векторы
- Предлагать, что интересно запрограммировать