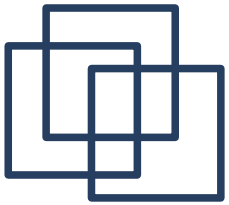
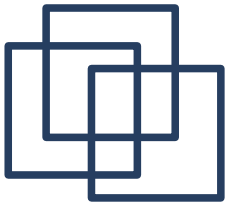


Кафедральный практикум 5 семестр

Часть 2. Язык Scheme (окончание)

http://sp.cmc.msu.ru/~kornykhin/fp/slides/part2-3.pdf



План

Часть 1. Введение.

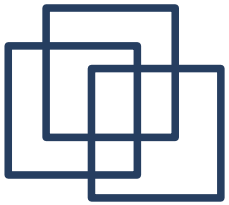
1) Организационные вопросы
2) Функциональный стиль программирования

Часть 2. Язык программирования Scheme.

1) Быстрый старт
2) **Более внимательный взгляд на Scheme**

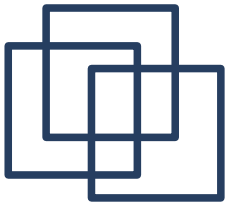
Часть 3. Функции высшего порядка. «Векторное» мышление.

Часть 4. Теоретический фундамент ФП.



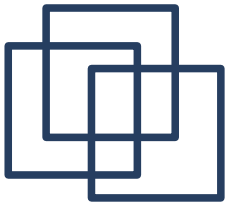
Умникам и умницам

- Doctor @ http://sp.cmc.msu.ru/~kornykhin



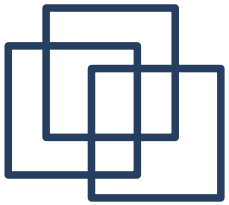
Lambda

- для определения *локальных* функций
- для записи «формул как фактических параметров» (пример: filter в лабиринте)
    - признак функциональности ЯП — наличие такой возможности
- для задания порядка вычислений:
  (+ (a) (b)) → ((lambda (x) (+ (a) x)) (b))
- для записи «эволюционирующей функциональности» (ИИ)



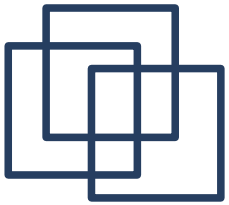
define и lambda

- define функций определяется при помощи lambda:
- (define (f x) expr) эквивалентно
  (define f (lambda (x) expr))



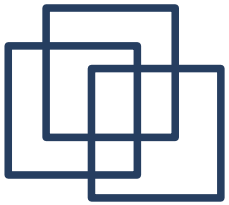
Факториал с lambda

- Естественное, но не всегда эффективное вычисление факториала:
- ```
  (define (factorial n)
      (if (= n 0)
          1
          (* n (factorial (- n 1)))))
  ```
  ```



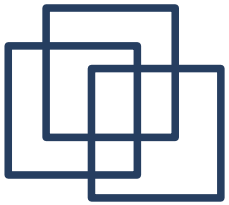
Факториал с lambda

- Неестественное, но эффективное вычисление факториала:
- ```
(define (factorialr n accum)
    (if (= n 0)
        accum
        (factorialr (- n 1) (* n accum))))
```
- ```
(define (factorial n)
     (factorialr n 1))
```
- Как соединить преимущества обеих реализаций?



Факториал с lambda

- каков бы ни был φ, должно быть верно, что
  - φ(n!) ≡ φ(n * (n-1)!) ≡ ψ((n-1)!),
  - где ψ(x) ≡ φ(n*x)
- ```
  (define (factorialr φ n)
      (if (= n 0)
          (φ 1)
          (let ((ψ (lambda (x) (φ (* n x))))
             (factorialr ψ (- n 1)))))
  ```
- ```
  (define (factorial n)
      (factorialr n (lambda (x) x)))
  ```

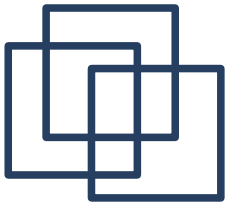


Факториал с lambda

- или в более человеческом виде:

```
(define (factorialr n доп-работа)
  (if (= n 0)
    (доп-работа 1)
    (let ((доп-работа-для-n-1 (lambda (n-1!)
                (доп-работа (* n n-1!))))
          (factorialr (- n 1) доп-работа-для-
n-1))))

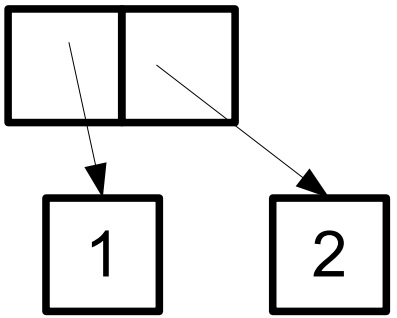
(define (factorial n)
   (factorialr n (lambda (x) x)))
```



Точечные пары

- cons, car, cdr
- список — набор точечных пар
- из точечных пар можно строить деревья

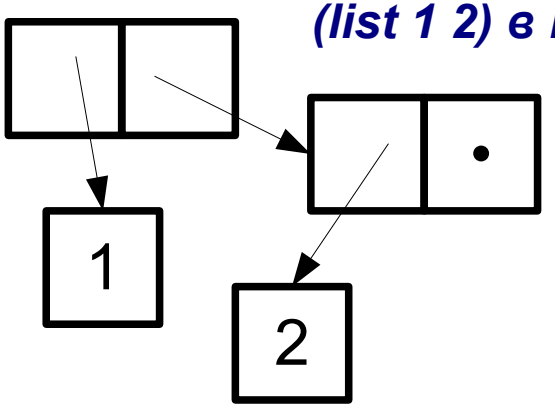
(cons 1 2) в памяти

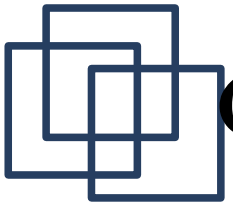


(cons 1 2) в программе

(1 . 2)

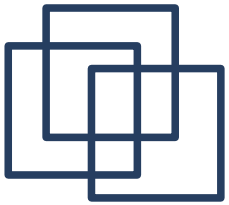
(list 1 2) в памяти





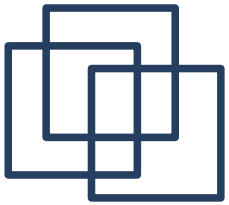
Функции с переменным числом параметров

- Определение: (define (f . x) expr)
- Определение: (lambda x expr)
- Вариант вызова: (f 1) → x = (1)
- Вариант вызова: (f 1 2 3) → x = (1 2 3)
- Вариант вызова: (f . 1) → x = 1
- (lambda (x y . others) expr) — у этой функции не менее двух аргументов (x, y — первые два аргумента, остальные аргументы - others)



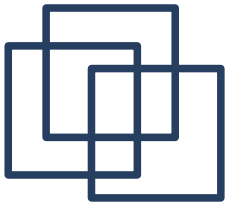
Локальные функции

- Всё ли тут в порядке?
-
```
(define const 1)
(define (f x)
    (define fx (+ const (factorial x)))
    (define (factorial n) (factorial-tail n) )
    (define const (+ x 10))
    (define (factorial-tail n accum)
        (if (= n 0)
            (+ const accum)
            (factorial-tail (- n 1) (* n accum))))
    ....some code with using of factorial ...
```



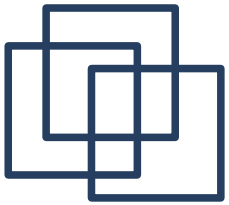
Разные let

- let определяет ("связывает") новое имя со значением
- Разные let отличаются правилом связывания имени и значение (имя переопределяется, если уже было определено)
- (**let** ((*x expr1*) (*y expr2*)) *expr*) - правило: «параллельное» вычисление *expr1* и *expr2*
- (**let*** ((*x expr1*) (*y expr2*)) *expr*) - правило: «последовательное» вычисление: *expr1*, затем *expr2*, в *expr2* можно использовать *x*



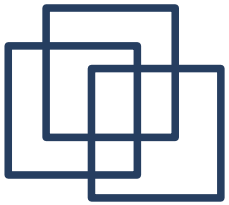
Разные let

- (**letrec** ((x expr1) (y expr2)) expr) - вычислить "параллельно" expr1 и expr2 (их вычисление **не должно использовать значение** x и y), вычислить expr
- (**letrec*** ((x expr1) (y expr2)) expr) - вычислить "последовательно" expr1 и expr2 (вычисление expr1 не должно использовать значение x и y, вычисление expr2 может использовать **значение** x), вычислить expr
- (**let** LOOP ((x initexpr)) bodyexpr) - в bodyexpr можно использовать (LOOP expr)



Корректно ли это?

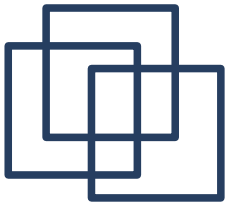
- (letrec ((x (y 1))
  (y (lambda (b) (+ b 1)))) x)
  - *поведение не определено*
- (letrec* ((x (y 1))
  (y (lambda (b) (+ b 1)))) x)
  - ошибка времени исполнения
- (letrec* ((y (lambda (b) (+ b 1)))
  (x (y a))) x)
  - корректное выражение
- Совет: аккуратно вызывать процедуры в letrec и letrec* (lambda ничего не вызывает)



Каков результат?

```
(let ((a 1))
 (define (f x)
  (let ((b (+ a x))
        (a 5))
    (+ a b) ) )
  (f 10))
```

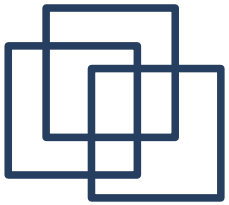
```
(let ((a 1))
 (define (f x)
  (let* ((b (+ a x))
         (a 5))
    (+ a b) ) )
  (f 10))
```



Каков результат?

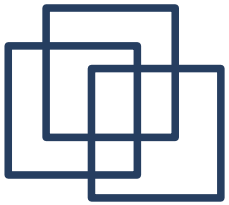
```
(let ((a 1))
 (define (f x)
  (letrec
     ((b (+ a x))
      (a 5))
    (+ a b) ) )
  (f 10))
```

```
(let ((a 1))
 (define (f x)
  (define b (+ a x))
  (define a 5)
    (+ a b) )
  (f 10))
```



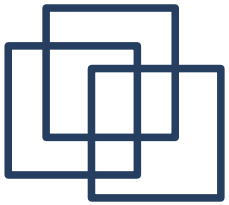
eq, eqv, equal

- виды данных:
  - скалярные (boolean, имена-данные, ( ))
  - «длинные» числа (number, character)
  - структуры данных (пары, списки, строки)
  - процедуры-данные
  - порты
- boolean?, pair?, symbol?, number?, char?, string?, port?, procedure?
- = char=? eq? eqv? equal?
- memq? memv? member? - хвост с первого вхождения



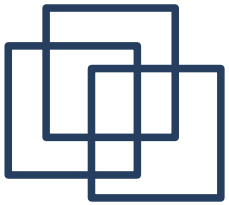
#f

- Удобно использовать не только как логическую ложь, но и как результат вычисления, когда оно невозможно (не выполнено *предусловие* функции)
  - нельзя построить путь в лабиринте, если старт и финиш не связаны
  - нельзя вычислить максимум в пустом списке
- (cond expr1 expr2 … exprN) — если "вычислилось" expr1, то оно есть результат, иначе если "вычислилось" expr2, то оно, иначе … результат exprN
- Пример: (cond (f x) (display «error»))



Преобразования

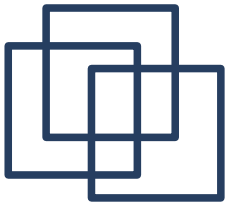
- number->string, string->number
- symbol->string, string->symbol
- list->string, string->list *(list from charaters!)*
- integer->char, char->integer *(<exact integer in [0,#x10FFFF], not in [#xD800,#xDFFF]>)*



Ввод/вывод

- (current-input-port) → port
- (current-output-port) → port
- (current-error-port) → port

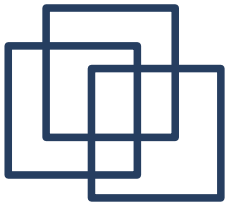
- (open-output-file path) → port, (open-input-file path) → port
- (read-line port) → string, (read port) → expr
- (print/write/display expr port)
- (eof-object? port) → boolean
- (close-output-port port), (close-input-port port)



Доктор

- Doctor @ http://sp.cmc.msu.ru/~kornykhin
- Сеанс у психотерапевта:

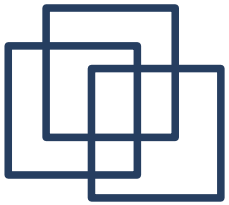
```
> (visit-doctor 'Bill)
(HELLO, BILL)
(WHAT SEEMS TO BE THE TROUBLE?)
**(everyone hates me)
(WHY DO YOU SAY EVERYONE HATES YOU)
**(whenever i enter a room no one will look me in the eye)
(MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS)
**(they laugh at me behind my back)
(YOU SEEM TO THINK THEY LAUGH AT YOU BEHIND
YOUR BACK)
. . .
```



Доктор

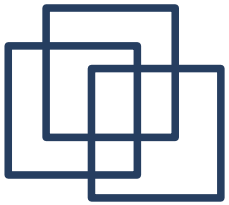
- Doctor @ http://sp.cmc.msu.ru/~kornykhin
- Сеанс у психотерапевта:

. . .
**(they get together and talk about how they hate me)
(MANY OF MY PATIENTS HAVE TOLD ME THE SAME THING)
**(even my mother hates me)
(YOU FEEL THAT EVEN YOUR MOTHER HATES YOU)
**(even when i was a baby she said i was ugly and stupid)
(PLEASE CONTINUE)
**(she hit me all the time)
(WHY DO YOU BELIEVE SHE HIT YOU ALL THE TIME)



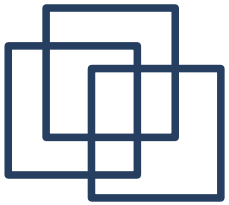
Доктор

- Организовать сеанс с пользователем
- Организовать ответы на реплики пользователя:
  - однозначный ответ: "please go on", "many people have the same sorts of feelings"
  - повтор реплики с заменой "я" на "Вы", "мой" на "Ваш"



Доктор

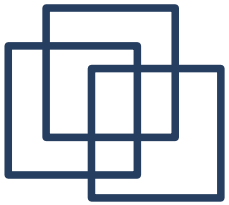
- Добавить новые стратегии формирования ответа доктора:
  - "EARLIER YOU SAID THAT" далее одна из предыдущих реплик пациента с заменой "меня" на "Вас" и подобные
  - организовать сеанс с несколькими пользователями (сначала разговор с одним, он уходит, потом разговор с другим)
  - если в реплике встречается определенное слово, доктор на него отвечает заранее заготовленной фразой ("I am often **depressed**" ⅛ "When you feel depressed, go out for ice cream"); продвинутый вариант: вставлять это слово в ответ доктора в указанном во фразе месте



Возвращаясь к идеям

- Программирование без заботы о том, что память закончится
- Программирование значениями без адреса
- Программирование без модификации и удаления данных (только создание/определение новых данных)

- Прежде, чем браться за серьезное кодирование, стоит отладить логику работы системы



Домашнее задание

- одна задача в ejudge : http://earth.ispras.ru
- дорешивание задач первой домашней работы в ejudge (с штрафом)
- В решении запрещается использовать
  - set! и его аналоги
  - векторы
- Контрольная работа на следующем семинаре
- Предлагать, что интересно запрограммировать (спецкурс Дениса Турдакова)