

# ФУНКЦИИ В SCHEME: АРГУМЕНТЫ, РЕКУРСИЯ, ЗАМЫКАНИЯ

Выполнил Гребенкин Егор Дмитриевич  
Группа 5030102/20202

# Процедуры как значения (first-class functions)

В Scheme функции (процедуры):

- ▶ можно присваивать переменным
- ▶ передавать как аргументы
- ▶ возвращать из других функций
- ▶ хранить в структурах данных

Это основа функционального стиля и DSL

## Объявление функций: define для функции

```
(define (add a b)  
  (+ a b))
```

Эквивалентно:

```
(define add (lambda (a b) (+ a b)))
```

# lambda

lambda создаёт функцию:

```
(define inc (lambda (x) (+ x 1)))
```

# Входные данные: аргументы

В Scheme обычно говорят так:

- ▶ аргументы передаются как значения
- ▶ но если значение — это "ссылка на объект в куче" (например список), то можно наблюдать эффекты мутации

# Передача по значению/ссылке: практика

Практически:

- ▶ числа/символы часто ведут себя как "значения" (не мутируются)
- ▶ списки/векторы могут мутироваться, и тогда "две переменные" могут видеть изменения

## Вариативные аргументы (rest args)

В Scheme можно принимать "остаток" аргументов как список:

```
(define (f a b . rest) ...)
```

rest — обычный список, с ним работают через car/cdr/null? или функции над списками

# Выходные данные

В минимальном Scheme функция возвращает одно значение — последнее вычисленное выражение

Некоторые реализации поддерживают multiple values (не ядро для всех стандартов), но базово предполагаем одно значение

# Рекурсия

Scheme традиционно использует рекурсию как универсальный механизм повторения

# Хвостовая рекурсия (Tail Call Optimization, TCO)

Многие реализации Scheme гарантируют оптимизацию хвостовых вызовов (в стандартах это важный пункт)

Это значит:

- ▶ хвостовая рекурсия может работать как цикл без роста стека

# Замыкания

Замыкание — это функция + захваченное окружение (значения внешних переменных)

# Замыкания: возможности

Замыкания позволяют:

- ▶ создавать функции-конфигураторы (make-adder)
- ▶ хранить состояние без глобальных переменных (make-counter)
- ▶ делать message-passing "объекты"

# Демонстрационные программы

Аргументы, variadic, высший порядок

Файл: 03/src/functions-demo.scm

## Вывод functions-demo.scm

```
add 2 3: 5
mul 6 7: 42
sum2+ 1 2: 3
sum2+ 1 2 3 4 5: 15
apply-twice inc 10: 12
map inc '(1 2 3): (2 3 4)
add10 7: 17
```

# Рекурсия и хвостовая рекурсия

Файл: 03/src/recursion-demo.scm

## Вывод recursion-demo.scm

```
fact 5: 120  
fact-tail 5: 120  
sum-to 10: 55
```

# Замыкания: счётчик и "аккаунт"

Файл: 03/src/closures-demo.scm

## Вывод closures-demo.scm

```
c1(): 1  
c1(): 2  
c2(): 101  
c1(): 3  
c2(): 102  
balance: 50  
deposit 20: 70  
withdraw 10: 60
```

# Как запустить в DrRacket (Racket Desktop)

- 1) DrRacket
- 2) File -> Open... -> .scm 03/src
- 3) Language -> Choose Language... ( Scheme- )
- 4) Run

## Альтернатива: запуск через Racket из терминала (опционально)

```
racket functions-demo.scm  
racket recursion-demo.scm  
racket closures-demo.scm
```

# Литература и материалы

- ▶ См. references.md
- ▶ SICP: замыкания, окружения, рекурсия vs итерация
- ▶ Dybvig: процедуры как значения, хвостовые вызовы

# Арность, "сигнатуры" и стиль API

Scheme динамически типизирован, но "форма" функции всё равно важна:

- ▶ арность: сколько аргументов ожидается
- ▶ наличие rest-аргументов (`. rest`)
- ▶ соглашения об именах и типах

# Практика для читаемости

- ▶ маленькие функции с понятными именами
- ▶ явные проверки входных данных в публичных функциях (через `error/assert`)
- ▶ встраивание соглашений в тесты/демо

apply и "список аргументов"

Иногда аргументы уже лежат в списке. Тогда:

```
(apply + '(1 2 3 4)) ; => 10
```

# Когда полезен apply

Это особенно полезно, когда:

- ▶ пишете обобщающие функции
- ▶ строите DSL, где "вызов" хранится как данные

# Каррирование, частичное применение, композиция

Scheme не навязывает каррирование, но оно легко реализуется функциями высшего порядка

## Частичное применение

```
(define (partial2 f a)  
  (lambda (b) (f a b)))
```

# Композиция

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

Эти кирпичики сильно упрощают построение пайплайнов преобразований

Демо: 03/src/hof-demo.scm

# Рекурсия как "шаблон" обработки коллекций

Типовой шаблон по списку:

```
(define (process xs)
  (if (null? xs)
      <base>
      <combine (car xs) (process (cdr xs))>))
```

# Рекурсия vs функциональный стиль

С точки зрения FP это почти всегда можно переписать через `map/filter/fold`

## "Замыкание как объект": message passing

Мы уже показали "аккаунт" через замыкание. Это важно как мост к ООП:

- ▶ состояние приватно
- ▶ доступ идёт через "сообщения" (msg)
- ▶ можно моделировать интерфейсы без классов

# Плюсы и минусы замыканий как объектов

## Плюсы:

- ▶ работает в любом Scheme (без классов)
- ▶ легко тестировать

## Минусы:

- ▶ нет наследования "из коробки"
- ▶ сложнее организовать общий протокол сообщений

# Передача "по значению/по ссылке": практическая формулировка

Корректнее думать так:

- ▶ переменная указывает на значение
- ▶ для составных структур значение — это объект, доступный по ссылке
- ▶ если вы мутируете объект, изменения видят все, кто держит ссылку

# Почему это важно

Это объясняет, почему:

- ▶ списки/векторы ведут себя как "shared mutable state"
- ▶ а числа/символы обычно безопасны для свободного копирования

# Демонстрационная программа №4: НОФ-паттерны

Файл: 03/src/hof-demo.scm

## Вывод hof-demo.scm

```
inc-then-sq 4: 25  
add10 7: 17  
filter odd? '(1 2 3 4 5): (1 3 5)  
foldl + 0 '(1 2 3 4): 10
```

# Реализационные "плюшки": keyword-аргументы в Racket

Это не ядро стандарта Scheme, но в Racket (и некоторых других реализациях) встречаются keyword-аргументы

Идея: у части аргументов есть имена-ключи (удобно для функций с множеством настроек)

## Важно для курса

- ▶ в примерах мы опираемся на переносимые идеи (R5RS/R7RS-стиль)
- ▶ реализационные расширения используем аккуратно и подписываем, когда это "только конкретная реализация" (например Racket)

## Мини-FAQ: частые вопросы про функции

Почему  $(f)$  и  $f$  — разное?

$f$  — значение (процедура).  $(f)$  — вызов процедуры без аргументов

Почему "замыкание хранит состояние"?

Потому что `lambda` захватывает переменные из внешнего `let`, а `set!` меняет их в этом окружении

Почему рекурсия иногда "падает по стеку"?

Потому что не вся рекурсия хвостовая. Делайте хвостовую рекурсию (accumulator) или используйте `fold`

# Возврат значений: "последнее выражение"

Базовое правило:

- ▶ результат функции — значение последнего выражения в теле

```
(define (f x)  
  (+ x 1)) ; (+ x 1)
```

## Multiple values (опционально)

Некоторые реализации поддерживают "множественные значения" (multiple values), но эта тема "поверх ядра" и в курсе будет упоминаться только обзорно, чтобы не ломать переносимость примеров