ПРОДВИНУТЫЕ ТЕМЫ SCHEME: FP/OO/МАКРОСЫ/КОНКУРЕНТНОСТЬ/VM/ПАМЯТЬ

Выполнил Гребенкин Егор Дмитриевич
Группа 5030102/20202

Функциональное программирование (FP) в Scheme

Scheme исторически связан с FP, потому что:

- функции — значения (first-class)
- удобно работать со списками
- рекурсия и хвостовая рекурсия позволяют выражать "циклы"функционально
- замыкания позволяют строить абстракции

Практические FP-паттерны

- map, filter (если есть), fold
- композиция функций
- неизменяемые структуры (или дисциплина "не мутировать")
- функции высшего порядка для повторного использования

FP-пайплайн как "данные -> преобразования -> результат"

Типовой подход:

- берём коллекцию
- фильтруем
- отображаем (map)
- сворачиваем (fold) в ответ

Преимущества FP-пайплайна

Это часто заменяет "циклы с кучей переменных"и даёт:

- меньше состояния
- проще тестировать
- легче менять шаги пайплайна

Демо: 06/src/fp-demo.scm

Объектно-ориентированное программирование (ООП)

В "чистом"Scheme ООП не обязано входить в стандарт, но:

- его можно моделировать message-passing через замыкания (см. 03/src/closures-demo.scm)
- в Racket есть встроенная объектная система (class/object%)

ООП в Racket: классы

Racket даёт:

- классы и объекты
- методы и наследование
- интерфейсы/миксины (как расширение модели)
- наследование/обобщение — средствами системы объектов

Демо: 06/src/oop-demo.scm

Когда ООП полезнее, чем чистые замыкания

ОО-подход часто выигрывает, когда:

- много типов сущностей с похожими операциями
- хочется "общих методов"и полиморфизма
- удобно описывать протокол "сделай X для объекта Y"

Реализация ООП в Scheme

В Scheme эти задачи можно решать:

- либо через замыкания/message passing (простые объекты)
- либо через объектную систему реализации (в Racket — классы)

"ZZZZZZ-ориентированное программирование"

Это пункт-шутка, но смысл полезный:
в Lisp-подобных языках легко делать DSL (domain-specific language) и "свои мини-языки"

Ключевой инструмент — макросы

Макросы:

- расширяют синтаксис
- позволяют писать "код, который пишет код"
- при грамотной реализации — гигиеничные (не ломают области видимости)

Демо макросов

Демо: 06/src/macros-demo.scm:

- макрос unless как синтаксический сахар
- мини-DSL rule как пример "языка правил"

Специфические конструкции языка: макросы

syntax-rules
syntax-rules — декларативный способ описывать переписывание синтаксических форм
Преимущества:

- проще и безопаснее для учебных DSL
- обычно обеспечивает гигиену

Как читать syntax-rules: "шаблон -> подстановка"

Макрос обычно выглядит как набор правил:

- слева — шаблон вызова
- справа — во что он разворачивается

Напоминание про ...

expr ... означает "ноль или больше выражений"

Пример макроса when

```
(define-syntax when
  (syntax-rules ()
    ((when test expr ...)
     (if test (begin expr ...) #t))))
```

Это почти "сахар"для if + begin

Гигиена (очень важно)

Гигиеничный макрос:

- не ломает лексические области видимости
- не "подсовывает"переменные так, чтобы они конфликтовали с именами пользователя

Именно поэтому syntax-rules часто рекомендуют как первый шаг

Конкурентность / параллелизм

Важно различать:

- конкурентность: структура программы позволяет выполнять задачи "вперемешку"(threads, async)
- параллелизм: задачи реально выполняются одновременно на разных ядрах/процессорах

Конкурентность в Scheme

В Scheme это сильно зависит от реализации:

- треды могут быть "настоящими"или кооперативными
- может быть ограничение, влияющее на параллельность
- могут существовать futures, event loop, акторные модели — как библиотечные решения

Потоки в Racket

В Racket есть потоки (threads) и синхронизация (mutex/semaphore и т.п.)
Демо: 06/src/threads-demo.scm

Shared mutable state — главный источник боли

Если несколько потоков разделяют мутабельные данные, появляются:

- race conditions
- "плавающие"баги
- зависимость от планировщика

Минимальная защита — mutex (блокировка)
Демо: 06/src/mutex-demo.scm

Функциональный стиль как упрощение конкурентности

Если вы строите программу так, что:

- данные "не мутируются"(или мутируются локально)
- функции не имеют побочных эффектов

то многие гонки исчезают сами собой

Функциональный стиль: вывод

Это не "серебряная пуля но сильная инженерная эвристика

Виртуальная машина (VM) и как "это запускается"

На высоком уровне реализации Scheme могут:

- интерпретировать AST напрямую
- компилировать в байткод/VM-код
- компилировать в машинный код (JIT/AOT)

Racket и модель выполнения

Racket использует свою реализацию рантайма/VM (внутренние детали — реализационно-зависимы)
Практический вывод для разработчика:

- "производительность"и "параллельность"в Scheme — это всегда связка язык + реализация + режим

Что можно оптимизировать на уровне кода

Даже без глубоких знаний VM полезно помнить:

- хвостовая рекурсия (TCO) помогает писать циклы без роста стека
- лишние аллокации списков могут замедлять (лишние cons)
- иногда лучше использовать векторы/порты/буферы (в зависимости от задачи)

Управление памятью

Чаще всего:

- память в куче
- автоматическая сборка мусора (GC)
- нет "владения"как обязательной концепции

Влияние GC на стиль

Это влияет на стиль:

- легче писать высокоуровневые абстракции
- но важно понимать мутабельность (shared mutable state) в конкурентных программах

GC-мышление (практично)

GC упрощает жизнь, но полезно помнить:

- короткоживущие объекты (временные списки) обычно "дешевле чем кажется
- но миллион временных объектов в горячем цикле всё равно будет стоить времени
- поэтому "алгоритм важнее а структура данных важна для производительности

Демонстрационные программы

Макросы и мини-DSL
Файл: 06/src/macros-demo.scm

Вывод macros-demo.scm

```
unless ran
x: 10
normalize-hello "HELLO": "hello"
normalize-hello "Hi": "Hi"
```

ООП (Racket: классы)

Файл: 06/src/oop-demo.scm

Вывод oop-demo.scm

```
point-x: 3
point-y: 4
dist2: 25
```

Потоки (threads)

Файл: 06/src/threads-demo.scm

Вывод threads-demo.scm

```
t1 done, acc=...
t2 done, acc=...
joined r1: ...
joined r2: ...
```

(порядок строк может отличаться)

FP-пайплайн (filter/map/fold)

Файл: 06/src/fp-demo.scm

Вывод fp-demo.scm

```
xs: (1 2 3 4 5 6)
odds: (1 3 5)
squares: (1 9 25)
sum: 35
```

Mutex и общий счётчик

Файл: 06/src/mutex-demo.scm

Вывод mutex-demo.scm

```
counter (expected 100000): 100000
```

Как запустить в DrRacket (Racket Desktop)

1) DrRacket
2) File -> Open... -> .scm 06/src
3) Language -> Choose Language...
4) Run

Альтернатива: запуск через Racket из терминала (опционально)

```
racket macros-demo.scm
racket oop-demo.scm
racket threads-demo.scm
racket fp-demo.scm
racket mutex-demo.scm
```

Частые грабли: Макросы

"Двойное вычисление"аргументов
Плохой макрос (идея):

- он может подставить expr дважды
- и если expr имеет побочный эффект, вы получите неожиданный результат

Правило для макросов

В макросах думайте о том, сколько раз вычисляется каждое подвыражение после разворачивания

Частые грабли: Потоки

Общий set! без mutex
Если несколько потоков делают:

- (set! counter (+ counter 1))

без синхронизации, итог почти наверняка будет "меньше ожидаемого"из-за гонок

Частые грабли: FP-стиль

"Случайная мутация"
Если вы договорились "не мутировать списки то:

- избегайте set-car!/set-cdr!
- возвращайте новые структуры (через cons, map, filter)

Как выбирать подход: FP

FP:

- когда важна прозрачность преобразований
- когда легко выразить задачу через map/filter/fold
- когда хочется уменьшить состояние (особенно в конкурентности)

Как выбирать подход: OO

OO (Racket: классы):

- когда есть много сущностей с общими операциями
- когда полиморфизм/диспетчеризация по типам упрощают архитектуру

Как выбирать подход: DSL

DSL (макросы):

- когда доменная задача повторяется шаблонно
- когда синтаксический сахар реально сокращает и проясняет код
- когда вы готовы поддерживать "внутренний язык"как API

Continuations (call/cc)

В некоторых курсах Scheme отдельно изучают continuations (call/cc):

- это мощный механизм управления потоком (ранние выходы, backtracking, корутины)
- но он сложен и сильно зависит от контекста и стиля кода

Continuations: для этого курса

Для этого набора презентаций достаточно знать, что такие конструкции существуют и почему Scheme часто используется для демонстрации "нестандартного"управления потоком

Литература и материалы

- См. references.md
- SICP: макросы/абстракции/интерпретация (концептуально)
- Dybvig: язык Scheme и реализация на уровне идей
- Racket: классы, threads, macro system