ОШИБКИ, ОТЛАДКА, МОДУЛИ, ВВОД-ВЫВОД В SCHEME (RACKET/DRRACKET)

Выполнил Гребенкин Егор Дмитриевич
Группа 5030102/20202

Обработка ошибок

В "классическом"Scheme ядро языка минимально, а детали ошибок часто зависят от реализации
Тем не менее, базовые идеи одни и те же:

- есть способ сигнализировать ошибку (error)
- есть способ перехватить и обработать ошибку (в Racket — with-handlers)
- практично иметь assert-проверки для инвариантов

Исключения: сигнализация ошибки

```
(error "message" extra-data...)
```

Исключения: перехват

В Racket есть with-handlers:

```
(with-handlers ([exn:fail? (lambda (e) ..... )])
  ...-  ... )
```

Assert и "встроенные средства отладки"

В минимальном Scheme нет единого стандартного assert, но его легко сделать самим:

```
(define (assert pred msg)
  (if (not pred) (error msg) #t))
```

Это подходит как "легковесная отладка особенно в учебных примерах

Управление сложностью: пространства имён и модули

Почему модули важны

- разносить код по файлам
- контролировать экспорт (публичный API)
- избегать конфликтов имён
- улучшать тестируемость и повторное использование

Модули в Racket

В Racket модуль обычно начинается с #lang и управляет экспортом через provide
Пример:

- файл: 05/src/enpl/math-utils.scm
- подключение из другого файла: (require "enpl/math-utils.scm")

Экспорт модуля

```
#lang racket
(provide square cube)
```

Импорт модуля

```
(require "enpl/math-utils.scm")
```

Практика: в DrRacket удобнее держать файлы рядом и подключать их по относительному пути

Стандартная библиотека ввода-вывода (обзор)

В Scheme обычно используются порты (ports):

- stdin/stdout/stderr — порты
- файлы открываются как порты
- можно читать/писать через процедуры display, write, read, read-line и т.п.

Порты в Racket

В Racket удобно:

- call-with-input-file
- call-with-output-file

Демонстрационные программы

Модули: импорт/экспорт
Файлы:

- 05/src/enpl/math-utils.scm
- 05/src/module-main.scm

Вывод module-main.scm

```
square 5: 25
cube 3: 27
```

Ошибки + assert + перехват

Файл: 05/src/errors-demo.scm

Вывод errors-demo.scm

```
assert #t: #t
safe-div 10 2: 5
caught exception: misc-error args=(...)
try safe-div 1 0: error
```

I/O: запись и чтение файла

Файл: 05/src/io-demo.scm

Вывод io-demo.scm

```
read lines: ("line1\n" "line2\n")
```

Как запустить в DrRacket (Racket Desktop)

1) DrRacket
2) File -> Open... -> .scm 05/src
3) Language -> Choose Language...
4) Run

Альтернатива: запуск через Racket из терминала (опционально)

```
racket module-main.scm
racket errors-demo.scm
racket io-demo.scm
```

Литература и материалы

- См. references.md
- Racket: Modules, Exceptions, Ports and I/O
- Scheme reports: базовая семантика, минимальное ядро (модули/исключения часто за пределами ядра)

Ошибки как часть дизайна API

Полезная привычка: заранее решить, что делает функция при ошибке:

- кидает исключение (error)
- возвращает специальное значение (#f, 'error, '(error ...))
- возвращает пару (ok/value) и (error/reason) — как протокол

Для учебных примеров

- инварианты и "не должно случиться"→ error
- "не нашлось/не распарсилось"→ #f или 'none

catch: что реально приходит в handler

В обработчик catch попадают:

- key — "класс"исключения (часто символ)
- args — дополнительные данные

Что можно делать с with-handlers

На практике вы можете:

- логировать объект исключения e
- возвращать запасное значение
- пробрасывать исключение дальше (rethrow) — уже как продвинутый вариант

Отладка без "магии"

Мини-набор:

- display — печать "для человека"
- write — печать "для воспроизводимого чтения"
- в Racket есть pretty-print — печать структур красиво

Идиома отладки

```
(display "debug: ")
(write some-structure)
(newline)
```

Это быстро и достаточно для 80% учебной отладки

Модули глубже: почему путь файла важен

В Racket удобно выстраивать структуру проекта по папкам и подключать код через require:

- (require "enpl/math-utils.scm")

Преимущества соответствия пути и имени

Это даёт:

- предсказуемую структуру проекта
- простую навигацию
- проще переиспользовать код и собирать "публичный API"через provide

Полезная команда запуска (терминал, опционально)

```
racket module-main.scm
```

Порты: унификация ввода-вывода

В Scheme "ввод-вывод"обобщён:

- файл — это порт
- строка может стать портом
- stdin/stdout/stderr — тоже порты

Полезные функции для портов

Отсюда многие полезные функции:

- call-with-input-file
- call-with-output-file
- call-with-input-string
- call-with-output-string

Демонстрационная программа №4: сериализация

Файл: 05/src/read-write-demo.scm

Вывод read-write-demo.scm

```
serialized: "(user (name . \"Alice\") (age . 20) (tags \"fp\" \"lisp\"))"
parsed: (user (name . "Alice") (age . 20) (tags "fp" "lisp"))
equal? parsed data: #t
```

Принцип Lisp-мира

Это показывает важный принцип Lisp-мира:

- S-выражения — удобный формат данных
- read/write дают "round-trip"без отдельного парсера

Паттерн "try": обёртка над with-handlers

Часто хочется написать:

- "выполни, а если упало — верни #f (или 'error)"

Реализация try->false

Это удобно оборачивать:

```
(define (try->false thunk)
  (with-handlers ([exn:fail? (lambda (e) #f)])
    (thunk)))
```

Использование try->false

Тогда можно писать:

```
(try->false (lambda () (string->number "123"))) ; => 123
(try->false (lambda () (error "boom"))) ; => #f
```

Когда использовать try->false

Такой стиль помогает "изолировать"места, где ошибки допустимы, от мест, где ошибка — это баг

I/O глубже: явное открытие/закрытие портов

Вы уже видели call-with-input-file / call-with-output-file — они удобны тем, что сами закрывают порт

Когда нужны явные open/close

Иногда нужно явно:

- открыть порт
- прочитать кусками
- закрыть

Тогда используют:

- open-input-file, open-output-file
- close-port

Практическое правило

- если можно — используйте call-with-* (меньше утечек ресурсов)
- явные open/close — только когда есть причина