

Scheme

Scheme

- Функциональный язык
- Мы реализуем лишь его подмножество
- Пишем интерпретатор
- Попробовать: установите `mit-scheme` из `apt` или `brew`

Scheme: примеры выражений

- Есть числа, `boolean` (`#t` или `#f`), *символы* и *пары*
- Пара: два выражения в скобках, разделенные точкой

Примеры пар: `(1 . 2)`, `(1 . #t)`, `(lol . kek)`, `((1 . 2) . 3)`

- Выражения можно вычислять

`1` \Rightarrow `1`, `(+ 1 2)` \Rightarrow `3`, `(+ 1 (- 3 2))` \Rightarrow `2`

Scheme: пары и списки

- Существует особая запись `()` для пустого списка
- Список выражается как пара из первого элемента и списка остальных

```
(1 . (2 . (3 . ())))
```

```
pair => pair => pair => ()
```

|

|

|

1

2

3

- Сокращенная запись - `(1 2 3)`, это синтаксический сахар
- Это не все, читайте README, там описано подробно

Scheme: интерпретатор

- Как python, принимает строку, возвращает строку или ничего
- На самом деле содержит слои абстракции
- Стадии обработки:
 - токенизация
 - парсинг
 - вычисление
 - сериализация

Scheme: организация

- Задача поделена на 5 последовательных частей
 - `tokenizer, parser` по 2 балла: подготовительные этапы, простые
 - `basic`, 3 балла: реализация простых функций, но довольно много писать
 - `advanced`, 2 балла: переменные, скоупы, лямбды, об этом дальше
 - `tidy`, 1 балл: реализация сборщика мусора
- Читайте README, они подробные, сегодня только основные идеи
- Много кода, порядка 1.5-2 тысяч строк, лучше не откладывать
- Почти полная свобода в архитектуре, организации кода, паттернах

Токенайзер

- Зачем нужен:
 - Нормализация данных: удаление лишних пробелов, унарных плюсов перед числами
 - Упрощение дальнейшей логики парсинга, она работает уже с токенами
- Вход: поток символов, `std::istream`
- Выход: поток токенов, неявно через метод `Next()`
- Разделяем чтение из потока и получение текущего символа:
 - `void Next();` читает токен
 - `Token GetToken();` возвращает последний прочитанный токен, не меняет состояние

Токенайзер: пример

- Пусть пользователь ввел строку `' (+ 4 +5)`.
- Из нее получится такая последовательность токенов:

`Quote OpenParen Symbol (+) Const (4) Const (5) CloseParen`

- То есть, были проигнорированы последовательности пробелов и унарные плюсы. Строка `' (+ 4 5)` дала бы такую же токенизацию.

Парсер

- Зачем нужен:
 - Преобразование потока токенов в структуру, удобную для вычисления
 - Вообще говоря может быть устроен крайне сложно
 - В Scheme простая грамматика, можно строить AST налету
- Вход: поток токенов из токенайзера
- Выход: AST, синтаксическое дерево
- AST: структура данных, однозначно задающая выражение, ей оперирует вычислитель

Парсер: пример AST

- Пусть пользователь ввел строку `(+ 1 2)`.
- Из нее получится такая последовательность токенов

```
Quote OpenParen Symbol(+) Const(1) Const(2) CloseParen
```

- После парсинга получится AST, аналогичное `(+ . (1 . (2 . ())))`:

```
Cell { first = Symbol(+);
```

```
    second = Cell { first = Const(1);
```

```
        second = Cell { first = Const(2);
```

```
            second = nullptr } } }
```

Парсер: как писать

- Достаточно двух функций

```
std::shared_ptr<Object> Read(Tokenizer* tokenizer);
```

```
std::shared_ptr<Object> ReadList(Tokenizer* tokenizer);
```

- В терминах этих функций парсинг `(- 3 (+ 1 2) 4)` будет таким:

1. ReadList

1.1. Read => `-`

1.2. Read => `3`

1.3. ReadList

1.3.1. Read => `+`

1.3.2. Read => `1`

1.3.3. Read => `2`

1.4. Read => `4`

Вычисление

- Вход: результат парсинга, AST
- Выход: результат вычисления, **тоже AST**
- Не смешивайте слои абстракции
- **Сериализация ответа в строчку - не часть процедуры вычисления!**
- Удобно все объекты унаследовать от интерфейса `Object`
- Подумайте, какие еще действия присущи всем объектам, и добавьте в `Object` соответствующие методы. Какие-то из них сейчас обсудим

Вычисление: как писать

- Число и boolean **вычисляются** в себя

`1` \Rightarrow `1`, `#t` \Rightarrow `#t`

- Как **вычисляется** пара?

`(+ 1 2)` \Rightarrow `3`, `(+ 1 2 3 4)` \Rightarrow `10`

- Во что удобно вычислять символ?
- Если у каждого объекта есть **семантика вычисления**, то как лучше делать ее переопределение?

Вычисление: как писать

- Вычисление должно быть рекурсивным

`(+ 1 (+ 2 3))` \Rightarrow `(+ 1 5)` \Rightarrow `6`

- Как в итоге будет устроено вычисление пары:
 1. Получить по левой части **функцию**
 2. Вычислить правую часть
 3. Применить функцию к результату вычисления правой части
- Еще нужна валидация аргументов на каждом шаге. В тестах вы найдете много интересных корнер-кейсов ;)

Вычисление: как писать

- Всего нужно написать 34 функции и особые формы
- Пишите хелперы
- Полезно обобщить свойства функций и написать абстрактный код
 - Функции-свертки (допускающие 0 аргументов и не допускающие)
 - Монотонные функции
 - Функции вида `IsExpectedType`
 - ...
 - Может быть полезно: <https://en.cppreference.com/w/cpp/utility/functional>

Вычисление: переменные, лямбды и скоупы

- Можно создавать переменные (`define`) и изменять уже созданные (`set`)
- Можно определять лямбда-функции, два вида синтаксиса:

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

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

- Лямбды могут захватывать контекст, как в C++
- Лямбды могут рекурсивно вызывать себя

```
> (define (fib x) (if (< x 3) 1 (+ (fib (- x 1)) (fib (- x 2))  
)))
```


Вычисление: переменные, лямбды и скоупы

- `define` может быть и внутри лямбды, такие переменные снаружи не видны
- Вам потребуется ввести понятие скоупа
- Всегда есть глобальный скоуп, в нем живут билтины
- Если переменной нет в локальном скоупе, нужно искать ее в скоупах выше, то есть скоупы иерархичны

Работа с памятью

- В заготовке используются `shared_ptr`, `enable_shared_from_this`
- Что может пойти не так?
- `scheme-basic`: не должно быть проблем с `shared_ptr`
- `scheme-advanced`: будут утечки, но мы их не проверяем
- `scheme-tidy`: написание Mark-and-Sweep сборщика мусора
- Читайте README `scheme-tidy`, алгоритм там описан