Scheme

Scheme

- Функциональный язык
- Попробовать: установите `mit-scheme` из apt или brew
- Мы пишем интерпретатор к нему
  - парсинг в синтаксическое дерево
  - вычисление выражений
  - переменные, лямбда-функции с захватом контекста

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)`

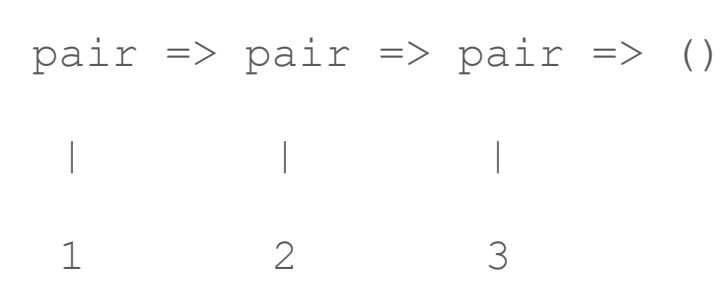
- Выражения можно вычислять

  `1` => `1`, `(+ 1 2)` => `3`, `(+ 1 (- 3 2))` => `2`

Scheme: пары и списки

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

```
(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 ответа

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, не с char-ами
- Вход: поток символов, `std::istream`
- Выход: поток токенов через вызовы метода `Next()`

Токенайзер

- Что будем хранить в токенайзере, что будет его состоянием?
- Разделяем **чтение из потока** и **получение текущего символа**:
  - `void Next();` читает токен, меняет состояние
  - `Token GetToken();` возвращает последний прочитанный токен, не меняет состояние
- Что такое `Token`?

```
using Token = std::variant<ConstantToken, BracketToken, QuoteToken>;
```

Почему именно так?

Токенайзер: пример

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

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

- Строка `'(+ 4    +5  )` дала бы такую же токенизацию. То есть, были проигнорированы последовательности пробелов и унарные плюсы. На последующих этапах про них не думаем

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
- Abstract syntax tree: 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. Если сделаете иначе, в advanced будете переписывать :)

Вычисление: как писать

- Удобно все объекты (числа, пары, символы, …) унаследовать от интерфейса `Object`
- Какие действия присущи всем объектам
  - Сериализация в строку
  - Клонирование
  - Вычисление
  - Что-то еще?
- Добавьте в `Object` соответствующие виртуальные методы

Вычисление: как писать

- Число и boolean **вычисляются** в себя

  `1 => 1, #t => #t`

- Как **вычисляется** пара?

  `(+ 1 2) => 3, (+ 1 2 3 4) => 10`

- Как бы вы написали функцию вычисления пары?

Вычисление: как писать

- Вычисление должно быть рекурсивным

  `(+ 1 (+ 2 3))` => `(+ 1 5)` => `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 builtins
- 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 сборщика мусора

Mark-and-Sweep: идеи

- Избавляемся от shared_ptr, делаем единое хранилище объектов
- Можно чистить все объекты в `~Interpreter()`? Или нет?
- Mark-and-Sweep: ищем недостижимые компоненты связности и удаляем целиком
- Вопросы
  - Недостижимые откуда?
  - Каким алгоритмом ищем?
  - Как удаляем недостижимые?

Mark-and-Sweep: иллюстрация