Scheme

Что такое scheme

- Язык программирования "без синтаксиса"

```
42 ; constant

(+ 1 2) ; expression

(define x 3) ; variable

(define (add x y) ; function
  (+ x y))
```

Почему scheme

- Хорошая тренировка в декомпозиции
- Нетривиальное использование наследования
- Хороший пример, как работать с исключениями

REPL

```
$ 1
> 1

$ (+ 1 2)
> 3
```

- REPL расшифровывается как Read, Eval, Print, Loop.
- `$` будем обозначать ввод пользователя
- `>` будем обозначать вывод интерпретатора

Разделение на Read, Eval и Print

- *Read* - считывает ввод пользователя
- *Eval* - производит вычисления
- *Print* - печатает результат назад

В языке `scheme` *Read* и *Eval* работают с одним типом данных.

```
std::shared_ptr<Object> Read();
std::shared_ptr<Object> Eval(std::shared_ptr<Object> ptr);
void Print(std::shared_ptr<Object> ptr);
```

** *не полные сигнатуры*

Что такое `Object`

```
class Object
    : public std::enable_shared_from_this<Object>
{
public:
    virtual ~Object() {}

    // Other methods....
};
```

Все объекты языка представляются наследниками типа `Object`.

Наследоваться от `enable_shared_from_this` не обязательно,
но может быть полезно.

Для управления памятью мы будем использовать `shared_ptr`.

Наследники `Object`

```
$ 1
$ a
$ (inc 1)
```

- `Number`. Обычные числа `int64_t`.
- `Symbol`. Имена в языке называются "символами".
- `Cell` - пара. Композитный тип.

Пустой список будем представлять как nullptr.

```
$ ()
```

Cell

```
class Cell : public Object {
private:
    std::shared_ptr<Object> first_, second_;
};
```

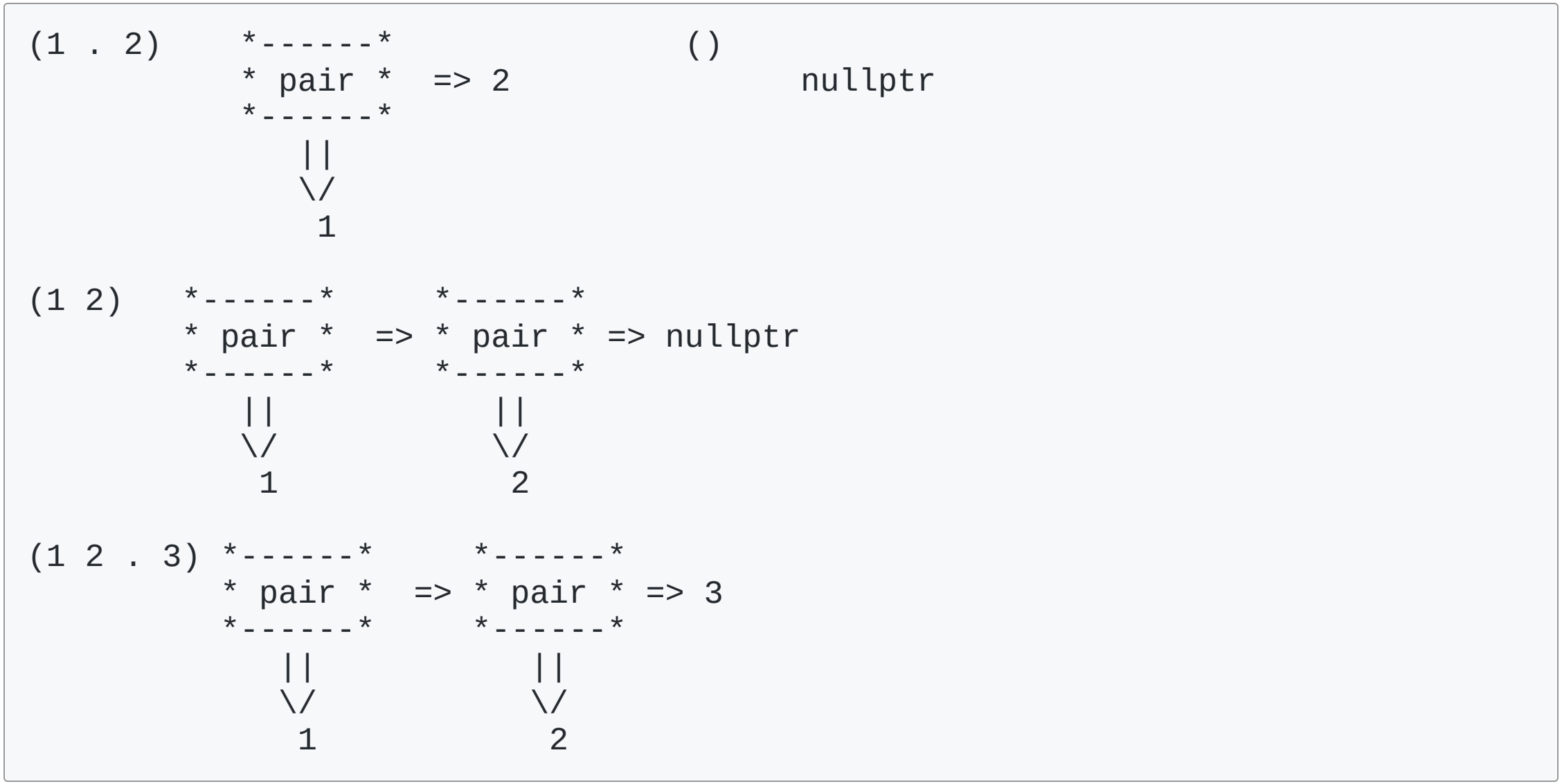
```
$ (1 . 2) ; Пара из двух чисел

$ () ; Нулевой указатель

$ (1 . (2 . (3 . ()))) ; Список
$ (1 2 3) ; Другой синтаксис для списка

$ (1 2 3 . 4) ; Список без nullptr в конце
```

Структура списков



Reader

Принимает на вход последовательность из `(`, `)`, `'`, `.`, чисел и символов.

Возвращает один `std::shared_ptr<Object>`.

```
(1 2 . 3) *------*     *------*
          * pair *  => * pair * => 3
          *------*     *------*
             ||           ||
             \/           \/
              1            2
```

Tokenizer

```
"(sum +2 . -3)"

{ "(", "sum", "+2", ".", "-3", ")" }
```

Читает байты из `std::istream*` и выдаёт поток токенов.

Токен будем хранить в `std::variant`.

```
struct SymbolToken {
    std::string name;
};

struct QuoteToken {
};

typedef std::variant<SymbolToken, QuoteToken>
    Token;
```

Eval

```
$ 1
> 1

$ (+ 1 2)
> 3

$ '(+ 1 2)
> (+ 1 2)

$ (define x 5)
> ()

$ x
> 5
```

Eval функций

```
$ (+ 1 (* 2 3))

~ (#<builtin-+> 1 6)

> 7
```

Функция сложения определена в C++

Символ `+` вычисляется в объект "функции".

Функция вычисляет все аргументы по обычным правилам. Затем, запускает `C++` логику.

Eval особых форм

```
$ (define x 1)
> ()

$ (quote (1 2 3))
> (1 2 3)
```

Символы `quote` и `define` вычисляются в объект "особой формы".

Аргумент передаётся в форму как есть, без вычисления.

Quote

```
$ '(+ 1 2)
> (+ 1 2)

$ (quote (+ 1 2))
> (+ 1 2)
```

`'` - это синтаксический сахар для `(quote ...)`.

Нужно переписывать одно в другое в `Reader`.

Scope

`Scope` хранит в себе имена.

Все встроенные имена определены в глобальном `Scope`.

`define` добавляет имя в `Scope`.

Вызов функции создаёт локальный `Scope`.

```
$ (define x 3)
$ (define (add a)
    (+ a x))

$ (add 1 2)
```

Lambda capture in C++

```
auto Range(int start) {
  return [start] () mutable {
    ++start;
    return start;
  };
}

void F() {
  auto r = Range(10);

  std::cout << r() << std::endl; // 11
  std::cout << r() << std::endl; // 12
}
```

Lambda capture

```
$ (define range
    (lambda (x)
      (lambda ()
        (set! x (+ x 1))
        x)))

$ (define my-range (range 10))

$ (my-range)
> 11

$ (my-range)
> 12
```

`Scope` могут быть сколь угодно вложенными.

Управление памятью

Мы будем использовать `shared_ptr`.

В наивной реализации будут утечки памяти, из-за циклов по `shared_ptr`.

Leak sanitizer отключен при тестировании на сервере, но есть бонус на 500 баллов, где нужно реализовать сборку мусора.

Приведение типов в вашем коде

```
template<class T>
std::shared_ptr<T> As(const std::shared_ptr<Object>& obj);

template<class T>
bool Is(const std::shared_ptr<Object>& obj);
```

Для тестов парсера, вам нужно определить пару хелперов `Is` и `As`. В отличии от использования `std::dynamic_pointer_cast` напрямую, вы можете задать нужную обработку ошибок.

Тестирование

```
struct SchemeTest {
    SchemeTest() {
        // Create new interpreter HERE.
    }

    // Implement following methods.
    void ExpectEq(std::string expr, std::string result);
};

TEST_CASE_METHOD(SchemeTest, "IntegersAreSelfEvaluating") {
    ExpectEq("4", "4");
}
```

- Тело `TEST_CASE_METHOD` выполняется внутри метода наследника от `SchemeTest`.
- `ExpectEq` вызывает ваш метод.
- Экземпляр `SchemeTest` создаётся на каждый тест.