Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Разработка интерпретатора подмножества языка Scheme

Студент ИУ9-71б _____ М. Ибрагимов
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ А.В. Синявин
(Подпись, дата) (И.О.Фамилия)

Москва

2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данной работы – создание интерпретатора подмножества языка программирования Scheme (R5RS) на языке программирования Rust.

Программа должна выполнять чтение файла, содержащего код на языке Scheme, а затем выполнять его.

Необходимо изучить существующие средства для анализа исходного кода программ и выполнения инструкций. Также необходимо изучить архитектуру построения современных интерпретаторов.

На основе полученных данных реализовать интерпретатор.

1. Обзор предметной области

Компилятор из языка *S* в язык *T* – это программа, транслирующая программу на языке *S* в программу на языке *T*.

Чаще всего, компилятор – это программа, переводящая код на высокоуровневом языке программирования в машинный код (ассемблер).

Интерпретатор для языка *S* – программа (схожа по структуре с компилятором), выполняющая анализ, обработку и тут же выполнение инструкций, написанных на языке *S*.

Как правило, интерпретатор состоит из трёх компонент:

- Лексический анализатор (лексер)
- Синтаксический анализатор (парсер)
- Обработчик инструкций

Две первые компоненты – называют фронтендом интерпретатора (так же и для компилятора). Фронтенд интерпретатора определяется его входным языком.

В данном случае, компоненты реализованы для обработки и выполнения инструкций подмножества языка Scheme.

В отличие от компилятора, после работы фронтенда, код не преобразуется в ассемблер (или в какое-либо промежуточное представление используемое, для дальнейшего преобразования в ассемблер), а для выполнения инструкций создается обработчик, который заранее выполняет те или иные простейшие инструкции.

2. Составные компоненты интерпретатора

2.1 Лексический анализатор

Лексический анализ – процесс аналитического разбора входной последовательности символов на распознанные группы – лексемы – с целью получения на выходе идентифицированных последовательностей, называемых токенами.

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Язык, а точнее, его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике, не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. Д.).

Цель такой конвертации обычно состоит в том, чтобы подготовить входную последовательность для другой программы, например для синтаксического анализатора.

Лексический анализатор (лексер) – программа, выполняющая лексический анализ.

Для создания лексического анализатора в ООП языках обычно создается главный класс Scanner, который является итератором по токенам. Его метод *next()* осуществляет лексический анализ и попутно ведет список комментариев (например, об ошибках).

Токен, также является классом (Token), который содержит в себе информацию о своем типе (идентификатор/число/ошибка) и позицию (Position).

Позиция – класс (Position), используемый для хранения координат токена.

Существуют инструменты, которые облегчают создания лексера, например программа Lex – генератор лексеров, на основе описанной лексической структуры.

2.2 Синтаксический анализатор

Синтаксический анализ – процесс сопоставления линейной последовательности токенов естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево).

Синтаксический анализатор (парсер) – программа, выполняющая синтаксический анализ.

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

Существует несколько типов парсеров:

- Нисходящий парсер – продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.

- Восходящий парсер – продукции восстанавливаются из правых частей, начиная с токенов и заканчивая стартовым символом.

Метод рекурсивного спуска – алгоритм нисходящего синтаксического анализа, реализуемый путем взаимного вызова процедур, где каждая процедура соответствует одному из правил КС-грамматики или БНФ. Применения правил последовательно, слева-направо поглощают токены, полученные от лексического анализатора. Это один из самых простых алгоритмов синтаксического анализа, подходящий для полностью ручной реализации.

БНФ (Бэкуса – Наура форма) – формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории.

РБНФ – Расширенная форма Бэкуса – Наура.

Так же как и для лексера, существуют программы, облегчающие программирование парсеров. Программа Yacc – генератор парсеров. Yacc генерирует парсер на основе аналитической грамматики, описанной в нотации БНФ.

2.3 Обработчик инструкций

Обработчик инструкций – это ядро интерпретатора. В текущем проекте обработчиком инструкции является подобие стековой машины.

Стековая машина – это режим вычислений, в котором исполнительное управление полностью поддерживается путем добавления, чтения и удаления в порядке LIFO (Last In First Out) – с помощью стека.

Обычно, для такой стековой машины определяют простейшие операции, например, арифметические. Сумма здесь — это результат двух считываний (и

удалений) верхних значений стека, их суммирование и добавление вычисленного обратно в стек.

Также, для более сложных реализаций, используется операция вызова процедур.

В интерпретаторе, обработчик инструкций – это последняя компонента, которая выполняется после парсера.

3. Реализация интерпретатора

3.1 Описание используемых инструментов

Важным этапом создания программы является проектирование и выбор используемых инструментов. Для реализации интерпретатора был выбран современный язык программирования Rust.

Rust – это мультипарадигмальный компилируемый язык с сильной статической типизацией, сочетает в себе парадигмы функционального и процедурного программирования с объектной системой, основанной на трейтах (более гибкие интерфейсы). Компилятором гарантируется безопасная работа с памятью, благодаря системе владений (Ownership), а также валидацией времен жизни ссылок.

Ключевые приоритеты языка – безопасность, скорость и параллелизм. Rust используется для системного программирования, в частности для создания компиляторов и интерпретаторов.

Единица компиляции Rust – крейт (crate).

В Rust используется модульная система, что позволяет удобно распределять код на файлы (отдельные компоненты). Изначально, нет соответствия между файловой системой и модульной системой проекта. Для этого используются механизмы включения модулей в корневой крейт. Благодаря этому модули могут циклически зависеть друг от друга (модуль A вызывает функции модуля B, а модуль B вызывает функции модуля A).

В дальнейшем, все конфликты, которые могут возникать при подключении сторонних зависимостей (крейтов), разрешаются еще на этапе компиляции, т.к. крейты, образуют ориентированный ациклический граф (результат топологической сортировки).

Также, концепт времен жизни ссылок и запрет на использование сырых указателей – это то, чего нет в других языках программирования. Благодаря этому исключается возможность разыменования нулевого указателя, а также использование ссылок, которые живут дольше, чем объект, на который они ссылаются.

Еще одним плюсом является большая редкость получения утечек памяти (возможны только при неправильном использовании умных указателей, например, при работе с графовыми структурами данных).

Другими, не менее важными плюсами выбора языка Rust – являются: zero-cost abstractions, вывод типов (Хиндли-Милнер), соответствия паттернам, встроенная система сборки Cargo и удобность тестирования.

3.2 Формальное описание языка Scheme

Для того чтобы создавать анализаторы, нужна формально описанная грамматика.

Грамматика Scheme в РБНФ:

Программа (program):

<program> → <form>*

<form> → <definition> | <expression>

Определения (definition):

<definition> → <variable definition>

| <syntax definition>

| (begin <definition>*)

| (let-syntax (<syntax binding>*) <definition>*)

| (letrec-syntax (<syntax binding>*) <definition>*)

| <derived definition>

<variable definition> → (define <variable> <expression>)

| (define (<variable> <variable>*) <body>)

| (define (<variable> <variable>* . <variable>) <body>)

<variable> → <identifier>

<body> → <definition>* <expression>+

<syntax definition> → (define-syntax <keyword> <transformer expression>)

<keyword> → <identifier>

Выражения (expressions):

<expression> → <constant>

| <variable>

| (quote <datum>) | ' <datum>

| (lambda <formals> <body>)

| (if <expression> <expression> <expression>) | (if <expression> <expression>)

| (set! <variable> <expression>)

| <application>

| (let-syntax (<syntax binding>*) <expression>+)

| (letrec-syntax (<syntax binding>*) <expression>+)

| <derived expression>

<constant> → <boolean> | <number> | <character> | <string>

<formals> → <variable> | (<variable>*) | (<variable>+ . <variable>)

<application> → (<expression> <expression>*)

Идентификаторы (Identifiers):

<identifier> → <initial> <subsequent>* | + | - | ...

<initial> → <letter> | ! | $ | % | & | * | / | : | < | = | > | ? | ~ | _ | ^

<subsequent> → <initial> | <digit> | . | + | -

<letter> → a | b | ... | z

<digit> → 0 | 1 | ... | 9

Типы данных (Data) (включены в идентификаторы)**:**

<datum> → <boolean> | <number> | <character> | <string> | <symbol> | <list> | <vector>

<boolean> → #t | #f

<number> → <uinteger r>

<character> → #\ <any character> | #\newline | #\space

<string> → " <string character>* "

<string character> → \" | \\ | <any character other than " or \>

<symbol> → <identifier>

<list> → (<datum>*) | (<datum>+ . <datum>) | <abbreviation>

Числа (Numbers):

<uinteger r> → <digit r>+ #*

<digit 10> → <digit>

Из представленной выше РБНФ, не обрабатывается часть грамматики из секции определений и выражений.

3.3 Реализация лексического анализатора

Лексический анализатор проектировался по схожей, принятой архитектуре лексических анализаторов.

Листинг 1 – перечисление Token

```
#[derive(Debug, Clone)]
pub enum Token {
    Identifier(String),
    Value(ScmValue),
    OpenParen,
    ClosingParen,
    Sentiel,
}
```

Так как Enum может содержать в себе перечисления, которые являются юнион/кортежными структурами, то удобно реализовать класс Token (этап лексера) как раз с помощью Enum.

Можно заметить, что Identifier и Value содержат в себе значения, которые можно доставать с помощью соответствия паттернам.

Identifier соответствует описанию идентификатора в РБНФ. Value – это любые значения, типов: integer, float, bool, char, String, Symbol, DotPair, Procedure, Nil.

OpenParen – означает открывающую скобку, ClosingParen – закрывающую скобку, а Sentiel – обозначение конца строки (вспомогательный, для того чтобы легче было проверять конец строки).

Для токена с помощью макроса derive, автоматически реализуются трейты Debug и Clone.

Также, была создана структура Lexer, которая содержит в себе итератор по символам входящей строки, динамический массив токенов, текущий символ и текущие координаты.

Листинг 2 – структура Lexer

```
pub(super) struct Lexer {
    chars: IntoIter<char>,
    tokens: Vec<Token>,
    current: Option<char>,
    line: u32,
    column: u32,
}
```

Для структуры Lexer был реализован метод *new()* – подобие конструктора, который принимает в себя ссылку на строковый срез (в Rust есть тип умный указатель – строка String и тип str – срез (используется ссылка - &str, т.к. str Unsized), но благодаря трейту Deref, тип String неявно приводится к типу &str, следовательно мы можем передавать в качестве аргумента в эту функцию как статические строки, так и строки хранящиеся в куче).

Метод *increment()* переходит на следующий элемент (изменяя координаты и вызывая метод *next()* на итераторе)

Основной метод – *run()*, в нем идет бесконечный цикл по входному потоку, и с помощью соответствия паттернов (pattern matching) можно делать вывод как обрабатывать входной поток.

Также, реализованы методы для разбора различных типов токена:

- *parse_identifier()*
- *parse_number()*
- *parse_boolean()*
- *parse_string()*
- *parse_char()*

И вспомогательный метод *parse_delimiter()*, для проверки на разделитель между токенами.

Листинг 3 – функция *run()*

```
pub(super) fn run(&mut self) -> Vec<Token> {
    self.increment();

    while let Some(c) = self.current {
        match c {
            ' ' | '\n' => {
                self.increment();
            }
            '(' => {
                self.tokens.push(Token::OpenParen);
                self.increment();
            }
            ')' => {
                self.tokens.push(Token::ClosingParen);
                self.increment();
            }
            '#' => {
                self.increment();
                match self.current {
                    Some('\'') => self.parse_char(),
                    _ => self.parse_boolean(),
                }
            }
            '0'..='9' => self.parse_number(),
            '\"' => self.parse_string(),
            '[' | ']' | '{' | '}' | '|' | '\\' => {
                panic!("Unexpected {} {}", self.line, self.column);
            }
            _ => self.parse_identifier(),
        }
    }

    self.tokens.push(Token::Sentiel);
    std::mem::take(&mut self.tokens)
}
```

К примеру, когда на вход подается символ от ‘0’ до ‘9’, то вызывается метод разбора числа.

Конструкция while let позволяет нам проходить бесконечный цикл, пока к нам на вход поступают символы.

3.4 Реализация синтаксического анализатора

В данной реализации статический анализатор не является как таковым статическим анализатором по определению, т.к. не возвращает AST (Abstract Syntax Tree).

Это связано с тем, что при создании проекта, для простоты тестирования интерпретатора сначала создавалась стековая машина, а уже в дальнейшем лексер и парсер. Получается, что парсер, по факту, напрямую работает со стековой машиной.

Для создания парсера также как и в лексере была создана структура Parser, которая содержит в себе динамический массив токенов и индекс текущего токена в массиве.

Листинг 4 – структура Parser

```
pub struct Parser {
    tokens: Vec<Token>,
    idx: usize,
}
```

Для структуры Parser также был создан метод *new()*, который принимает на вход Vec<Token>, полученный в результате работы лексера, idx изначально инициализируется нулем.

Главная логика определена в функции *parse()* и *parse_expr()*. Парсер начинает работу из первой функции, и каждый раз, когда он видит открывающую скобку, он вызывает вторую функцию, которая разбирает само выражение.

Например выражение (+ 1 2) будет неявно преобразовано в (apply + (list 1 2)).

Парсер напрямую кладет инструкции на стек.

Лямбды парсятся справа налево.

Код `(display (+ 1 2))` со стороны инструкций преобразуется в:

1. ProcCall(“Display”, 1)
2. ProcCall(“+”, 1)
3. Val(ScmValue::Integer(1)
4. Val(ScmValue::Integer(2)

3.5 Реализация обработчика инструкций

Обработчик инструкций — это совокупность модулей и функций, отвечающих за обработку и выполнение инструкций.

Как упоминалось в лексере (Листинг 1), токен бывает различных типов; эти типы описаны в перечислении ScmValue. Одно из перечислений - Procedure, содержит в себе ScmCallable (перечисление типов процедур).

Основа выполнения инструкций — это вызов процедур. Процедуры могут быть как встроенные, так и созданные в контексте программы.

Листинг 5 – Созданные вспомогательные типы

```
#[derive(Clone)]
pub enum ScmCallable {
    Builtin(fn(ctx: &mut ScmExecContext, args: &[ScmValue]) -> ScmValue),
    CustomProc(ScmProcedure),
}

#[derive(Clone)]
pub struct ScmProcedure {
    pub params: Vec<String>,
    pub instructions: Vec<ScmProcUnit>,
}

pub enum ScmProcUnit {
    Val(ScmValue),
    Variable(String),
    ProcCall(String, usize), // Name and args cnt
    Lambda { args: Vec<String>, units_cnt: usize },
    TrueBranch(usize),  // Skip size
    FalseBranch(usize), // Skip size
    Assign(String),     // Define and assign are same here
}
```

ScmProcUnit – перечисление, включающее в себя список встроенных операций стековой машины.

- Val – положить значение на стек.
- Var – считать переменную и положить на стек.
- ProcCall<name, acnt> - снять со стека cnt значений, передать их как аргументы функции; результат положить на стек.
- Lambda<params, icnt> - создать новую функцию с параметрами params, тело функции – следующие icnt инструкций (переносятся в новую функцию), дополнительно все известные переменные заменяются на значения. Полученная функция кладется на стек.
- Assign<name> - снять значение со стека, присвоить его переменной name.
- TrueBranch<skip> - снять значение со стека, если оно истинно – то ничего не делать, иначе – пропустить skip инструкций.
- FalseBranch<skip> - пропустить skip инструкций.

Для контекста создана отдельная структура ScmExecContext

В контексте программы можно хранить заранее созданные встроенные (built-in) процедуры (хранятся в виде указателей на функцию).

Также, контекст может пополняться новыми значениями, которые добавляются с помощью define.

Листинг 6 – Структура ScmExecContext

```
pub struct ScmExecContext {
    pub variables: VariablesSet<ScmValue>,
}
```

Листинг 7 – Built-in функция умножения “*”

```
pub const SCM_BUILTIN_MUL: ScmValue =
    ScmValue::Procedure(ScmCallable::Builtin(|_, args| -> ScmValue {
        if args.len() == 0 {
            return ScmValue::Integer(1);
        }
```

```
        if args.len() == 1 {
            if let ScmValue::Integer(n) = args[0] {
                return ScmValue::Integer(n);
            }
        }

        for arg in args.iter() {
            match *arg {
                ScmValue::Integer(_) => (),
                _ => panic!("Unsupported value"),
            }
        }
        let mut mul = 1;

        for arg in args {
            if let ScmValue::Integer(n) = arg {
                mul *= *n;
            }
        }

        ScmValue::Integer(mul)
    }));
```

Вызов процедуры осуществляется, в соответствии паттерну ProcCall: сначала обработчик пытается найти функцию в аргументах функции, затем в контексте.

В главном бинарном крейте программы (main.rs), обрабатываются аргументы командной строки (передаем название файла, где хранится программа).

В результате получился интерпретатор со следующим функционалом:

- Вызов процедур (встроенных либо кастомных)
- Определение переменных и процедур (define)
- Условный переход, причем ложная ветка не вычисляется
- Лямбды: создание процедуры как значения (замыкания)

Добавление built-in процедур в контекст происходит в функции *new()* для структуры ScmExecContext:

Листинг 8 – Добавление built-in процедур

```
pub fn new() -> ScmExecContext {
    let mut ctx = Self {
        variables: VariablesSet::new(),
    };

    ctx.add_or_assign_var("+", SCM_BUILTIN_ADD);
    ctx.add_or_assign_var("-", SCM_BUILTIN_SUB);
    ctx.add_or_assign_var("*", SCM_BUILTIN_MUL);
    ctx.add_or_assign_var("/", SCM_BUILTIN_DIV);
    ctx.add_or_assign_var("<", SCM_BUILTIN_LE);
    ctx.add_or_assign_var("=", SCM_BUILTIN_EQ);
    ctx.add_or_assign_var("newline", SCM_BUILTIN_NEWLINE);
    ctx.add_or_assign_var("display", SCM_BUILTIN_DISPLAY);
    ctx.add_or_assign_var("list", SCM_BUILTIN_LIST);
    ctx.add_or_assign_var("apply", SCM_BUILTIN_APPLY);
    ctx.add_or_assign_var("cons", SCM_BUILTIN_CONS);
    ctx.add_or_assign_var("car", SCM_BUILTIN_CAR);
    ctx.add_or_assign_var("cdr", SCM_BUILTIN_CDR);

    ctx
}
```

4. Тестирование

Для проверки работы на каждом этапе применялись различные способы тестирования:

- Юнит-тестирование
- Интеграционное тестирование

Юнит-тестирование применяется для создания небольших тестов над какой-либо отдельно взятой компонентой. Так, например, юнит-тесты отдельно писались для лексера, парсера и стековой машины.

Листинг 9 – Пример юнит-теста для лексера

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::fs;

    #[test]
    fn test() {
        let contents = "(+ 1 2).to_string()"
        let mut l = Lexer::new(&contents);

        let v = l.run();

        assert_eq!(v, vec![Token::OpenParen, Token::Identifier("+"),
              Token::Value(ScmValue::Integer(1)),
                     Token::Value(ScmValue::Integer(2),
                            Token::ClosingParen)])
    }
}
```

Также, были реализованы юнит-тесты для стековой машины. Тестировалась она отдельно, способом инициализации контекста и стека инструкций.

Листинг 10 – Тестирование стековой машины

```
#[cfg(test)]
mod tests {
    use super::*;
```

```
    #[test]
    fn test_sub() {
        let mut instr: Vec<ScmProcUnit> = Vec::new();

        instr.push(ScmProcUnit::ProcCall(SCM_BUILTIN_SUB.clone(), 2));
        instr.push(ScmProcUnit::Val(ScmValue::Integer(3)));
        instr.push(ScmProcUnit::Val(ScmValue::Integer(1)));

        let p = ScmProcedure {
            params: Vec::<String>::new(),
            instructions: instr,
        };

        let mut ctx = ScmExecContext::new();
        let ret = exec_callable(&mut ctx, callable.clone(), &Vec::new());

        assert!(ret, ScmValue::Integer(2));
    }
}
```

Интеграционные тесты используются для проверки корректности программы в целом. Для этого был создан файл test.scm, содержащий в себе код для проверки функционала:

Листинг 11 – Файл test.scm

```
(define (add x)
    (+ x 1))

(define (fact x)
    (if (< x 2)
        1
        (* (fact (- x 1)) x)))

(define
    (adder x)
    (lambda (y)
        (+ x y)))

(display (+ 1 2))
(newline)
(display ((adder 10) 15))
(newline)
(display (fact 5))
(newline)
(define (f +) (+ 2 1))
(display (f -))
```

Листинг 12 – Результат работы интерпретатора

```
ScmValue::Integer(3)
ScmValue::Integer(25)
ScmValue::Integer(120)
ScmValue::Integer(1)
```

С помощью тестов был продемонстрирован реализованный функционал интерпретатора.

Для запуска тестов, в репозитории проекта достаточно запустить команду `cargo test`

ЗАКЛЮЧЕНИЕ

Написание интерпретатора довольно нетривиальная задача, т.к. “правильный” подход к разработке (разрабатывать масштабируемый код, с грамотной архитектурой; писать оптимально, используя нужные алгоритмы по назначению и т.д.) требует опыт и знания в данной области.

По итогу работы, был создан интерпретатор для подмножества языка Scheme, была хорошо выстроена архитектура проекта – человеку, не работавшим с данным проектом, но имеющим опыт написания подобных программ, будет легко разобраться в проекте, и, например, добавить какую-либо новую функцию.

Также, из литературы были изучены принципы построения современных интерпретаторов.

Так как работа над проектом велась на относительно новом языке Rust, был также получен дополнительный опыт работы с языком; удобства языка были использованы по возможности.

Дальнейшее развитие проекта:

- В данном проекте много недоработок, поэтому, для начала, код требует рефакторинга и исправления мелких багов.
- Далее, в проект нужно добавить обработку бóльшего подмножества языка Scheme (в частности, поддержку разных типов чисел во всех арифметических операциях, begin и eval)
- На данный момент, код уже поддерживает интерактивный способ выполнения инструкций (REPL (read-eval-print-loop)), но нужно добавить функцию обработки консольных запросов. Это даст дополнительные возможности, например, для быстрой проверки какой-либо операции, не нужно будет создавать файл и запускать его.

- Интересным исследованием, также, являлось бы изменение парсера, чтобы он строил дерево разбора, применение оптимизаций и использование кодогенератора LLVM. Это, например, позволило бы писать кросс-платформенные приложения на Scheme при компиляции в WASM байт-код.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Robert Nystrom. Crafting Interpreters [Электронный ресурс]
URL: https://craftinginterpreters.com/ (дата обращения 08.03.2022).

2. Ахо А., Лам Моника С. Компиляторы. Принципы, технологии и инструментарий. М.: Вильямс, 2016, с. 351-259.

3. Rust reference [Электронный ресурс]
URL: https://doc.rust-lang.org/reference/ (дата обращения 08.03.2022).

4. Formal Syntax of Scheme [Электронный ресурс]
URL: https://www.scheme.com/tspl2d/grammar.html (дата обращения 08.03.2022).

5. М. Ибрагимов [Репозиторий]. – Режим доступа:
URL: https://github.com/curryrasul/scheme_interpreter