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:

«Интерпретатор языка Scheme»

Студент ИУ9-72Б _____ А.Г. Владиславов
(Группа) (Подпись, дата) (И.О. Фамилия)

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

2022 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данной работы - реализация большой части функций интерпретатора подмножества R^5RS языка Scheme.

Программа интерпретатора должна считывать из потока ввода или из текстового файла код на языке Scheme и выдавать результат работы данной программы.

Необходимо изучить спецификации языка для написания интерпретатора.

Также необходимо изучить инструменты для написания легко масштабируемого и переносимого под разные платформы программного кода, с целью дальнейшего развития интерпретатора.

На основе изученных данных необходимо написать интерпретатор.

1 Постановка задачи, обзор технологий

Задачей данной работы было выбрано написание интерпретатора языка Scheme — функционального языка программирования, одного из наиболее популярных диалектов Lisp. На кафедре ИУ9 данный язык изучается на первом семестре первого курса, так что базовое понимание уже присутствует

1.1 Обзор интерпретируемого языка

В рамках курсовой будет разобрана большая часть стандарта R^5RS языка Scheme, описанного в документе [1]. Из данного стандарта будет реализован основной набор функций языка, чтобы на нём была возможность запускать лабораторные работы, выполняемые на первом курсе. Реализованные функции языка были поделены на категории, представленнив в таблице 1.

Таблица 1 — Релизованные функции языка.

Категория	Функции, относящиеся к катеригии
Операции с числами	+, -, *, /, number?
Операции с логическими значениями	not, boolean?
Операции с символами	char?
Операции со строками	string?
Операции с символами	symbol?
Равенства и неравенства	eq?, eqv?, >, <
Операции с парами и списками	car, cdr, cons, list, pair?, null?, set-car!, set-cdr!,
Синтаксические операции	quote, quasiquote, define, lambda, if, set!, begin, cond, else
Системный интерфейс	load
Ввод	eof-object?, read
Вывод	display, write, newline, flush,
Операции с портами ввода/вывода	port?, input-port?, output-port?, textual-port?, binary-port?, current-input-port, current-output-port
Вычисление	eval, null-environment, scheme-report-environment, interaction-environment, current-environment, in-environment, make-environment, environment?
Функции управления	call-with-current-continuation, procedure?, apply

В листинге 1 представлено описание грамматики языка Scheme в форме Бэкуса-Наура

Листинг 1: Описание грамматики языка Scheme в форме Бэкуса-Наура

```
<program> ::= <expression>
<expression> ::= <constant> | <variable> | <procedure-call> |
    <conditional> | <lambda> | <definition>
<constant> ::= <number> | <boolean> | <string> | <char> | <symbol>
<number> ::= [0-9]+
<boolean> ::= #t | #f
<string> ::= ".*"
<char> ::= #\\.
<symbol> ::= [a-zA-Z]+
<variable> ::= [a-zA-Z]+
<procedure-call> ::= (<procedure> <expression>*)
<procedure> ::= + | - | * | / | number? | not | boolean? | char? |
    string? | symbol? | eq? | eqv? | > | < | car | cdr | cons | list |
    pair? | null? | set-car! | set-cdr! | quote | quasiquote | define |
    lambda | if | set! | begin | cond | else | load | eof-object? | read
    | display | write | newline | flush | port? | input-port? |
    output-port? | textual-port? | binary-port? | current-input-port |
    current-output-port | eval | null-environment |
    scheme-report-environment | interaction-environment |
    current-environment | in-environment | make-environment |
    environment? | call-with-current-continuation | procedure? | apply
<conditional> ::= (if <predicate> <expression> <expression>)
        | (cond (<predicate> <expression>)+ [else <expression>])
<predicate> ::= <expression>
<lambda> ::= (lambda (<variable>*) <expression>)
<definition> ::= (define <variable> <expression>)
```

1.2 Выбор языка для написания интерпретатора

Для реализации интерпретатора было желание выбрать популярный язык, программный код на котором на котором возможно собрать на большинстве операционных систем, настройка окружения не должна занимать много времени, была возможность легко расширять программу и впоследствии написать графический пользовательский интерфейс для взаимодействия не только через командную строку. К данному описанию подходит язык Kotlin [2] — современный язык, разрабатываемый компанией JetBrains. Данный язык широко используется при разработке мобильных приложений. Помимо JVM, данный язык имеет LLVM и JS бэкенды компилятора, что значительно расширяет области применения, благодаря чему, при определенных модификациях программного кода (избавления от зависимостей Java) интерпретатор можно использовать не только для на пер-

сональном компьютере, но и на мобильных устройствах [3], на сервере [4] и в веб-страницах браузера [5].

2 Проектирование программного комплекса

Программа должна представлять из себя интерфейс командной строки. На вход пользователь вводит программный код на языке Scheme, на выход получает результат интерпретации введённого кода.

Программа состоит лексического анализатора, разбивающего входной текст на лексемы, синтаксического анализатора, составляющий дерево разбора и интерпретатора, обрабатывающего разобранное дерево и выполняющего код.

На рисунке 1 представлен пример разбора простого арифметического выражения на языке Scheme.

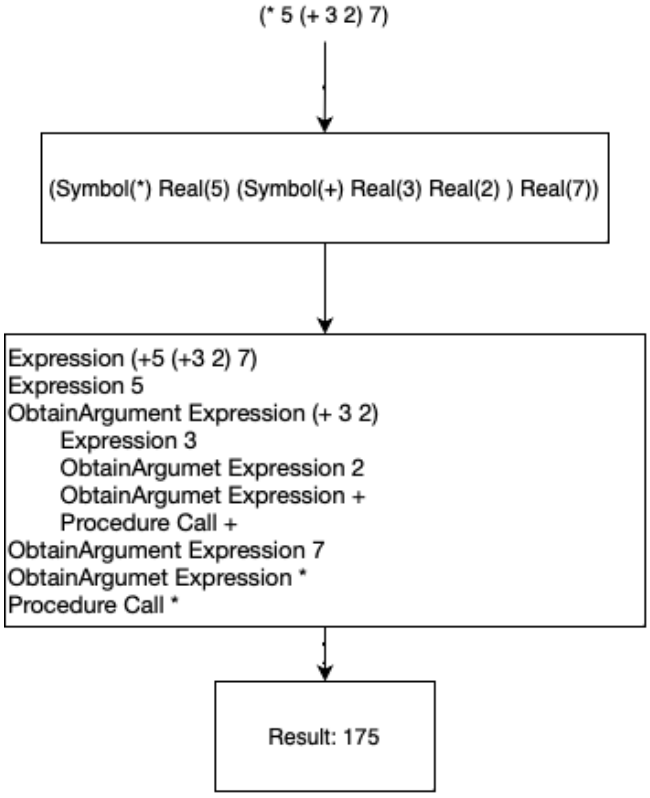

Рисунок 1 — Пример разбора арифметического выражения.

3 Разработка программного комплекса

3.1 Базовые элементы для сущностей

Для всех сущностей интерпретатора был создан интерфейс с набором методов, которые должны быть реализованы во всех наследниках интерфейса. Список методов представлен в листинге 2.

Листинг 2: Базовый интерфейс Entity

```
interface Entity : Serializable {

    fun eval(env: Environment, continuation: Continuation): Entity?

    fun analyze(env: Environment): Entity

    fun optimize(env: Environment): Entity

    fun write(out: PrintWriter)

    fun display(out: PrintWriter)

    fun toWriteFormat(): String

    override fun toString(): String
}
```

Интерфейс `Entity` реализуется в классе `AbstractEntity`, от которого наследуются практически все сущности в программном коде интерпретатора (помимо списков). В данном классе представлены базовые реализации методов интерфейса. Код данного класса представлен в листинге 3.

Также интерфейс `Entity` наследуется в интерфейсе: `List`. Данный интерфейс отвечает за последовательности сущностей и помимо `Entity` также реализует интерфейс `Iterable` для итерации по элементам списка, внутри себя имеет два поля: `car` и `cdr`, отвечающие за первый и последующие элменты списка. Код `List` представлен в листинге 4.

Листинг 3: Класс AbstractEntity

```
abstract class AbstractEntity : Entity {
    override fun eval(env: Environment, continuation: Continuation):
    Entity? = this

    override fun analyze(env: Environment): Entity = this

    override fun optimize(env: Environment): Entity = this

    override fun display(out: PrintWriter) {
        write(out)
    }

    override fun toWriteFormat(): String {
        val sw = StringWriter()
        display(PrintWriter(sw))
        return sw.toString()
    }

    override fun toString(): String {
        val sw = StringWriter()
        display(PrintWriter(sw))
        return sw.toString()
    }
}

```

Листинг 4: Интерфейс List

```
interface List : Entity, Iterable<Entity?> {
    var car: Entity
    var cdr: Entity
}
```

3.2 Действия над сущностями

Для действий был создан базовый класс `Action`. Данный класс выполняет различные действия с сущностями и имеет ссылку на следующее действие. Код данного класса представлен в листинге 5

Данный абстрактный класс наследуется семью другими для реализации действий.

Листинг 5: Класс Action

```
abstract class Action(
    val env: Environment?,
    var next: Action?,
) : Serializable {

    fun andThen(action: Action): Action {
        action.next = this.next
        this.next = action
        return action
    }

    abstract fun invoke(arg: Entity, cont: Continuation): Entity?

    protected fun interface Printer {
        fun print(port: OutputPort)
    }

    protected fun trace(doo: Printer, env: Environment?) {
        if (env?.interpreter?.traceEnabled == true) {
            val cout = env.out ?: return
            val actionName = this.javaClass.simpleName.replace("Action",
    "")
            cout.printf("%s ", actionName)
            doo.print(cout)
        }
    }
}

```

В данной релизации интерпретатора представлено 7 действий: `AssignmentAction`, `EvalAction`, `ExpressionAction`, `ExpressionInEnvAction`, `IfAction`, `ObtainArgumentAction`, `ProcedureCallAction`. В листинге 6 представлены реализации метода `invoke` для разных действий.

[!htb]

Листинг 6: Реализация метода `invoke` для различных действий

```
// Assignment --- операция присваивания
override fun invoke(arg: Entity, cont: Continuation): Entity? {
    cont.head = next
    env?.getLocation(symbol)!!.value = arg
```

```
5       trace({ out -> out.print("${symbol.toWriteFormat()} <-
        ${arg.toWriteFormat()}\n") }, env)
6       return Void.VALUE
7   }
8   // Eval --- операция вычисления
9   override fun invoke(arg: Entity, cont: Continuation): Entity? {
10      cont.head = next
11      trace({ out -> out.print("${arg.toWriteFormat()}\n") }, env)
12      return arg.eval(env ?: return null, cont)
13  }
14  // Expression --- вычисление выражения
15  override fun invoke(arg: Entity, cont: Continuation): Entity? {
16      cont.head = next
17      trace({ port -> port.printf("${expr.toWriteFormat()}\n") }, env)
18      return expr.eval(env ?: return null, cont)
19  }
20  // ExpressionInEnv --- вычисление выражения в определенном окружении
21  override fun invoke(arg: Entity, cont: Continuation): Entity? {
22      cont.head = next
23      if (arg !is Environment) {
24          throw Exception("Not an environment $arg")
25      }
26      val evalEnv = arg
27      expr = expr.analyze(evalEnv).optimize(evalEnv)
28      trace({ out -> out.print("${expr.toWriteFormat()}\n") }, env)
29      return expr.eval(evalEnv, cont)
30  }
31  // If --- условный оператор. Реализация также имеет поля consequent и
        alternate
32  override fun invoke(arg: Entity, cont: Continuation): Entity? {
33      cont.head = next
34      return if (arg != KSBoolean.FALSE) {
35          consequent.eval(env ?: throw Exception("Null env"), cont)
36      } else {
37          alternate.eval(env ?: throw Exception("Null env"), cont)
38      }
39  }
40  // ObtainArgument --- получение аргумента выражения
41  override fun invoke(arg: Entity, cont: Continuation): Entity? {
42      cont.head = next
```

```
43      argList.set(argumentIndex, arg)
44      trace({ out -> out.print("") }, env)
45      return arg
46  }
47  // ProcedureCall --- выполнение процедуры
48  override fun invoke(arg: Entity, cont: Continuation): Entity? {
49      cont.head = next
50      val operator: Procedure = arg as? Procedure ?: throw
        Exception("Operator not a procedure")
51      trace({ out -> out.print("${arg.toWriteFormat()}\n") }, env)
52      return operator.apply(argList.getArgs(), env!!, cont)
53  }
```

3.3 Примитивные процедуры

Для примитивных процедур был написан абстрактный класс `Primitives`, расположенный в пакете `lib`. Этот класс имеет поля:

- `name` — имя, как в программе;
- `definitionEnv` — окружение, в котором используется примитив;
- `keyword` — флаг, показывающий, является примитив частью синтаксиса;
- `minArgs` — минимальное число аргументов для использования примитива;
- `maxArgs` — максимальное число аргументов для использования примитива;
- `comment` — короткая заметка о примитиве;
- `documentation` — опциональное поле, более долгое объяснение, возможно с примером использования;

Также, поскольку большинство примитивов имеют ограниченное количество параметров для выполнения, этот класс имеет методы `apply0`, `apply1`, `apply2`, `apply3`, `applyN` — для выполнения процедуры с определённым количеством параметров.

3.4 Чтение и запись в консоль

На данном этапе программа должна представлять себа консольную утилиту, принимающую на вход набор команд на языке Scheme, интерпретирующую их и выдающую результат работы программы в вывод. Для ввода и вывода используются стандартные Java интерфейсы для чтения и записи в консоль (`java.io.Reader`, `java.io.PrintWriter`). Для более удобной работы с вводом и выводом были написаны классы - `InputPort` и `OutputPort`, наследуемые от базового абстрактного класса `Port`.

Для разбиения на токены используется класс `Reader`, который, в свою очередь использует `java.io.StreamTokenizer`. В листинге 7 представлен код инициализации класса `Reader`, в котором указываются интервалы для различных категорий токенов.

Листинг 7: Инициализация класса `Reader`

```
init {
    tokenizer.resetSyntax()
    tokenizer.lowerCaseMode(false)
    tokenizer.slashStarComments(false)
    tokenizer.commentChar(';'.code)
    tokenizer.quoteChar('\"'.code)
    tokenizer.whitespaceChars('\u0000'.code, '\u0020'.code)
    tokenizer.eolIsSignificant(false)
    tokenizer.wordChars('A'.code,'Z'.code) // A-Z
    tokenizer.wordChars('a'.code,'z'.code) // a-z
    tokenizer.wordChars('0'.code,'9'.code) // 0-9
    tokenizer.wordChars('\u00A1'.code,'\u00FF'.code) // Unicode latin-1
    supplement, symbols and letters
    tokenizer.wordChars('*'.code,'.'.code) // '*', '+', ',', '-', '.'
    tokenizer.wordChars('!'.code,'!'.code) // '!'
    tokenizer.wordChars('#'.code, '#'.code)
    tokenizer.wordChars('\\'.code,'\\'.code) // '\'
    tokenizer.wordChars('/'.code,'/'.code) // '/'
    tokenizer.wordChars('$'.code,'$'.code) // '$' $
    tokenizer.wordChars('_'.code,'_'.code) // '_'
    tokenizer.wordChars('<'.code,'@'.code) // '<', '=', '>', '?', '@'
}
```

Непосредственно чтение токенов происходи в методе `readToken`, представленном в листинге 8.

Листинг 8: Чтение токенов в классе `Reader`

```
private fun readToken(): String? {
    val c = tokenizer.nextToken()
    return when (tokenizer.ttype) {
        StreamTokenizer.TT_EOF -> null
        StreamTokenizer.TT_NUMBER -> tokenizer.nval.toString()
        StreamTokenizer.TT_WORD -> tokenizer.sval
        '\"'.code -> "\"${tokenizer.sval}"
        else -> c.toChar().toString()
    }.also {
        if (it != null) {
            Logger.log(Logger.Level.DEBUG, "TOKEN=$it")
        }
    }
}
```

Далее происходит разбор в методе `readObject`, представленном в листинге 9.

Листинг 9: Реализация метода `readObject`

```
private fun readObject(token: String): Entity? {
    if (token.isEmpty()) {
        return readObject()
    }
    when (token) {
        "(" -> {
            return readList(Pair(car = EmptyList.VALUE, cdr =
    EmptyList.VALUE))
        }
        "'" -> { return Pair(car = Symbol.QUOTE, Pair(car = readObject()
    ?: return null, cdr = EmptyList.VALUE)) }
        "`" -> { return Pair(car = Symbol.QUASIQUOTE, Pair(car =
    readObject() ?: return null, cdr = EmptyList.VALUE)) }
        ")" -> { throw Exception("Unexpected \")\"") }
        else -> { return readOthers(token) }
    }
}
```

3.5 Чтение из файла

Некоторые функции языка были взяты из документа prelude.scheme [6] — файла, разработанного для инициализации языка Scheme стандарта R4RS, разработанного в университете Массачусетса в Бостоне в 1990 году. В данном файле описаны многие функции, которые возможно реализовать на базе написанного в интерпретаторе кода. Данный файл загружается в интерпретатор при его инициализации, в функции, представленной в листинге 10. Файл `bootstrap.scm` находится в ресурсах, используемых программой.

Листинг 10: Чтение файла для инициализации некоторых функций

```
fun bootstrap() {
    if (!bootstrapped) {
        val bootstrap = InputPort(
            BufferedReader(
                InputStreamReader(
                    javaClass.getResourceAsStream("/bootstrap.scm") ?:
    throw Exception("file null")
                )
            )
        )
        load(bootstrap, reportEnv)
    }
}
```

Функция `load` считывает исходный код из файла, на данный момент в коде она использьуется только для инициализации, но на её основе была написана функция, используемая для тестирования, которая будет разобрана в соответствующем разделе.

3.6 Итоговая диаграмма классов

На рисунке 2 представлена итоговая диаграмма классов программного комплекса.

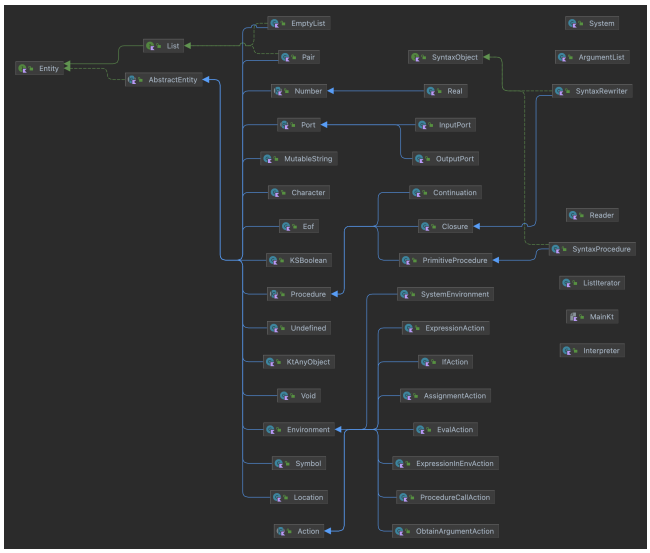

Рисунок 2 — Диаграмма классов программного комплекса.

4 Тестирование программного комплекса

4.1 Тесты

Для проверки работоспособности и корректности работы интерпретатора, был написан ряд unit-тестов, покрывающих основной функционал интерпретатора. Помимо основных проверок, таких как операции сложения, вычитания, умножения, деления, сравнения, операций со списками и так далее, был написан пример интерпретатора на языке Scheme, который должен быть разобран и запущен интерпретатором, написанным в рамках данной работы. Для проверки корректности тестов одни и те же фрагменты кода запускались "основным"интерпретатором и интерпретатором, используемом для тестирования. В листинге 11 представлен исходный код на языке Scheme, используемый для тестирования, в листинге 12 представлен код для запуска теста

Листинг 11: Интерпретатор для тестирования программного комплекса

```
(define (evlis exprs)
    (if (null? exprs)
        '()
        (cons (evaluate (car exprs)) (evlis (cdr exprs)) )))

(define (from-racket-bool x)
    (if x 't 'f))

(define (to-racket-bool x)
    (cond ((eq? x 't) #t)
          ((eq? x 'f) #f)
          (else (error 'to-racket-bool "not t or f"))))

(define (evaluate expr)
  (cond ((number? expr) expr)
    ((eq? (car expr) 'quote) (cadr expr))
    ((eq? (car expr) 'if) (if (to-racket-bool (evaluate (cadr expr)))
                              (evaluate (caddr expr))
                              (evaluate (cadddr expr))))
    (else (apply-fun (car expr) (evlis (cdr expr))))))

```

```
(define (apply-fun fun xs)
    (cond ((eq? fun '*) (* (car xs) (cadr xs)))
          ((eq? fun '+) (+ (car xs) (cadr xs)))
          ((eq? fun '-) (- (car xs) (cadr xs)))
          ((eq? fun '/) (/ (car xs) (cadr xs)))

          ((eq? fun 'car) (caar xs))
          ((eq? fun 'cdr) (cdar xs))
          ((eq? fun 'cons) (cons (car xs) (cadr xs)))

          ((eq? fun 'list) xs)
          ((eq? fun 'null?) (from-racket-bool (null? (car xs))))))
```

4.2 Автоматизация тестирования

Листинг 12: Код для тестирования интерпретатора

```
class SchemeInterpretTest {
    lateinit var interpreter: Interpreter

    @BeforeTest
    fun init() {
        interpreter = Interpreter.newInterpreter()
        val inputPort = InputPort(
            BufferedReader(
                InputStreamReader(

    javaClass.getResourceAsStream("/scheme/scheme-interpreter.scm") ?:
    throw Exception("file null")
                )
            )
        )
        interpreter.loadForTest(inputPort, interpreter.sessionEnv!!)
    }

    @Test
    fun runTest() {
        val expr = "(if (null? '(a b c)) 'a 'b)"
```

```
20          run(expr)
21      }
22
23      @Test
24      fun runTest2() {
25          val expr = "(* (+ 2 3) (- 10 2))"
26          run(expr)
27      }
28
29      @Test
30      fun runTest3() {
31          val expr = "(car (cdr (quote (a b c))))"
32          run(expr)
33      }
34
35      @Test
36      fun runTest4() {
37          val expr = "(cons (+ 1 7) '(b c d))"
38          run(expr)
39      }
40
41      @Test
42      fun runTest5() {
43          val expr = "(list (+ 3 9) (* 5 6))"
44          run(expr)
45      }
46
47      private fun run(expr: String) {
48          val expected = expr.byteInputStream()
49          val real = "(evaluate '$expr)".byteInputStream()
50          val expectedResult =
    interpreter.loadForTest(InputPort(InputStreamReader(expected)),
    interpreter.sessionEnv!!)
51          val realResult =
    interpreter.loadForTest(InputPort(InputStreamReader(real)),
    interpreter.sessionEnv!!)
52          assert(expectedResult == realResult)
53      }
54  } // $
```

Для поддержания стабильности работы исходного кода при добавлении нового кода было решено использовать Github Actions. Для общей стабильности при каждом коммите в ветку, из которой был создан `Pull request` запускается проверка сборки, чтобы в основную ветку не попал не собирающийся код. В листинге 13 представлен код, запускающий тесты для каждого Pull Request, созданного в одну из главных веток (`dev`, `master`). При не прохождении тестов, автор кода будет получит письмо от Github с информацией о причине неудачного коммита в ветку, а также будет заблокировано слияние в основную ветку. Пример подобного письма представлен на рисунке 3.

Листинг 13: Чтение файла для инициализации некоторых функций

```yaml
name: Pull Requests

on:
  pull_request:
    branches:
      - 'dev'
      - 'master'

jobs:
  test:
    name: "Run tests"
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: 'recursive'
      - name: Set up JDK 1.11
        uses: actions/setup-java@v2
        with:
          distribution: 'temurin'
          java-version: '11'
      - name: Run all tests
        run: ./gradlew :test
```

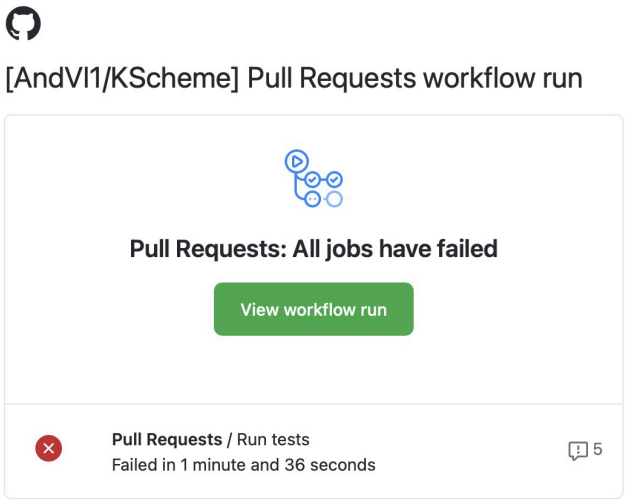

Рисунок 3 — Пример письма о непрохождении тестов.

ЗАКЛЮЧЕНИЕ

В ходе работы был создан интерпретатор подмножества R^5RS языка Scheme.

Программа выполняет чтение программного кода из файла или из командной строки.

Интерпретатор выполняет введённый программный код и выдаёт ответ пользвателю.

Программный код имеет много возможностей для модификации, от добавления нового функционала до написания приложений с графическим пользовательским интерфейсом для взаимодействия с интерпретатором и выполнения программного кода.

В рамках курсовой работы был написан интерпретатор языка Scheme на языке Kotlin/JVM. Интерпретатор умеет обрабатывать большую часть функций из перечисленных в стандарте R^5RS, тем не менее, есть достаточно большая возможность для расширения функционала. Также есть возможность в модификации программного кода таким образом, чтобы интерпретатор можно было использовать на различных платформах для выполнения программного кода.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Спецификация R^5RS [Электронный ресурс]. — URL: https://schemers.org/Documents/Standards/R5RS/.
2. Документация языка Kotlin [Электронный ресурс]. — URL: https://kotlinlang.org.
3. Использование Kotlin для мобильных приложений iOS/Android [Электронный ресурс]. — URL: https://kotlinlang.org/lp/mobile/.
4. Использование Kotlin для серверного кода [Электронный ресурс]. — URL: https://kotlinlang.org/docs/server-overview.html.
5. Использование Kotlin для Web приложений [Электронный ресурс]. — URL: https://kotlinlang.org/docs/js-overview.html.
6. UMB Scheme [Электронный ресурс]. — URL: https://linux.die.net/man/1/umb-scheme.