МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра вычислительные системы и технологии

ОТЧЕТ

по лабораторной работе №1

по дисциплине

Теория языков программирования и методов трансляции

РУКОВОДИТЕЛЬ:

_____ Кузнецов Г.Д.

СТУДЕНТ:

_____ Сапожников В.О.

19-В-1

Работа защищена «___» _____

С оценкой _____

Нижний Новгород 2022

Задание

Реализовать программу, выполняющая лексический анализ входного языка и создание таблицы лексем с указанием их типов. Программа должна выдавать сообщения о наличии во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Вариант №13

Входной язык содержит арифметические выражения, разделённые символом ≪;≫. Арифметические выражения состоят из идентификаторов, шестнадцатеричных чисел (последовательность цифр и символов a, b, c, d, e, f, которые начинаются с 0x. Например 0x00, 0xF0, 0xFF и т.д.), знака присваивания ≪=≫, знаков операций ≪+≫, ≪-≫, ≪*≫, ≪/≫ и круглых скобок.

Теоретическая часть

Трансляция имеет 4 фазы – лексика, синтаксис, семантика и генерация кода. В этой лабораторной работы мы пока остановимся на лексике.

Лексика языка программирования — это правила «правописания слов» программы, таких как идентификаторы, константы, служебные слова, комментарии. Лексический анализ разбивает текст программы на указанные элементы. Особенность любой лексики -ее элементы представляют собой регулярные линейные последовательности символов. Например, идентификатор — это произвольная последовательность букв, цифр и символа "_", начинающаяся с буквы или "_".

Лексический анализ. На этом этапе исходный текст программы «нарезается» на лексемы – последовательности входных символов (литер), допускающие альтернативу (выбор), повторение и взаимное пересечение в процессе распознавания не более чем на заданную глубину. Элементами лексики являются идентификаторы, константы, строки, комментарии, одно- и многосимвольные знаки операций и т.п.. Механизм распознавания базируется на простых системах распознавания, имеющих всего одну единицу памяти (текущее состояние) и табличное описание правил поведения (конечный автомат). Результат классификации лексем (классы лексем) является входной информацией для синтаксического анализатора (класс лексемы называется на входе синтаксического анализатора символом). Значения лексем поступают на вход
семантического анализатора в виде первичного заполнения семантических таблиц.

Практическая часть

Язык программирования: Kotlin

В ходе выполнения работы было создано три класса:

1. LexemeTable.kt – таблица лексем, содержит в себе записи о лексемах и их типах.
2. LexicalAnalyzer.kt – класс синтаксический анализатор. На основе допустимых элементов, операторов и разделителей выполняет синтаксический анализ входного файла.
3. TokenType. – перечисления типа лексем

Листинг

LexemeTable.kt

```
package Work1

import org.apache.commons.lang3.StringUtils

class LexemeTable {
    var content = mutableListOf<MutableList<String>>()

    fun add(lexeme: String, type: TokenType) {
        content.add(
            mutableListOf(lexeme, type.toString())
        )
    }

    fun print() {
        println("-".repeat(31))
        println("| " + String.format(
            "%s%s%s", StringUtils.center("Лексема", 14),
            StringUtils.center("|", 3),
            StringUtils.center("Тип", 10)) + " |")
        println("-".repeat(31))

        content.forEach {
            println("| " + String.format(
                "%s%s%s", StringUtils.center(it.first(), 14),
                StringUtils.center("|", 3),
                StringUtils.center(it.last(), 10)) + " |")
            println("-".repeat(31))
        }
    }
}
```

LexicalAnalyzer.kt

```
package Work1

/**
 * Лексический анализатор
 */
data class LexicalAnalyzer(
    val elements: List<Char>,
    val operators: List<Char>,
    val separators: List<Char>,
) {
    /**
     * Удаление многострочных комментариев, т.к. они игнорируются
```

```
     * @param text текст для анализа
     * @return текст, в котором удалены многострочные комментарии
     */
    private fun removeMultiLineComment(text: String): String {
        var result = text

        while ("/*" in result && "*/" in result) {
            result = result.removeRange(
                result.indexOf("/*"),
                result.indexOf("*/") + 2
            )
        }

        if ("*/" in result && "/*" !in result) throw Exception("Expecting an
element: " + text.indexOf("*/"))

        return result
    }


    /**
     * Удаление однострочного комментария
     * @param text текст для анализа
     * @return текст, в котором удалены однострочные комментарии
     */
    private fun removeSingleLineComment(text: String): String {
        var result = text

        while ("//" in result) {
            val start = result.indexOf("//")
            var end = 0
            for (i in start..text.length) {
                if (text[i] == '\n') {
                    end = i
                    break
                }
            }

            result = result.removeRange(start, end)
        }

        return result
    }

    /**
     * Выделение шестнадцатеричного числа
     * @param str строка, из которой выделяется число
     * @return 16-тиричное число
     */
    private fun selectHexNumber(str: String): String {
        var temp = ""

        for (ch in str) {
            if (ch in separators || ch == ';') break
            else if (ch in elements) temp += ch
            else {
                throw Exception("Unresolved reference: $str")
            }
        }

        return temp
    }
```

```
/**
 * Выделение идентификаторов
 * @param str строка, из которой выделяется идентификатор
 * @return идентификатор
 */
private fun selectIdentifier(str: String): String {
    var temp = ""

    for (ch in str) {
        if (ch in separators || ch == ';') break
        if (ch.toString().matches("[A-z0-9_]".toRegex())) temp += ch
        else {
            throw Exception("invalid character: ${str.indexOf(ch)}")
        }
    }

    return temp
}

/**
 * Выделение оператора
 * @param str строка, из которой выделяется оператор
 * @return оператор
 */
private fun selectOperator(str: String): String {
    var temp = ""

    if (str[0] in operators) temp += str[0]
    else {
        throw Exception("invalid character: ${str[0]}")
    }

    return temp
}

/**
 * Выделение присваивания
 * @param str строка, из которой выделяется присваивание
 * @return знак присваивания
 */
private fun selectAssigment(str: String): String {
    var temp = ""

    if (str[0] == '=') temp += str[0]
    else {
        throw Exception("invalid character: ${str[0]}")
    }

    return temp
}

/**
 * Ананлиз текста, построение таблицы лексем
 * @param text текст для анализа
 * @return таблицу лексем
 */
fun analyze(text: String): LexemeTable {
    val result = LexemeTable()
    var typeOfPreviousElem = TokenType.Assignment
    var lexeme: String

    //Удаление одно- и многострочных коментариев
    //И делим текст по пробелам
```

```
var strings = removeSingleLineComment(
        removeMultiLineComment(text)
    ).split("\\s".toRegex())

//Заменяем табуляцию одинарным пробелом
strings.forEach { it.replace("\t", " ") }
strings = strings.filter { it.isNotEmpty() }

//Анализ каждого выделенного элемента
strings.forEach {
    var temp = it

    //пока выделенный элемент не пуст
    while (temp.isNotEmpty()) {

        //удаляем пробел в начале элемента
        temp = temp.trimStart(' ')

        //Выделение разделителей
        if (temp[0] in separators)
        {
            lexeme = temp[0].toString()

            //Разделители (    )
            if (lexeme == "(") {
                if (")" !in text.substring(text.indexOf(lexeme),
                                text.length)) {
                    throw Exception("Expecting an element: " +
                                text.indexOf(temp))
                }
            }

            //Разделители {    }
            if (lexeme == "{") {
                if ("}" !in text.substring(text.indexOf(lexeme),
                        text.length)) {
                    throw Exception("Expecting an element: "
                        text.indexOf(temp))
                }
            }

            result.add(lexeme, TokenType.Separator)
            temp = temp.removePrefix(lexeme)
            typeOfPreviousElem = TokenType.Separator
        }

        //Выделение 16-тиричного числа
        if (temp.startsWith("0x")) {
            if (typeOfPreviousElem == TokenType.HexNumber) {
                throw Exception("Expecting an element: " +
                        text.indexOf(temp))
            }
            lexeme = selectHexNumber(temp)
            result.add(lexeme, TokenType.HexNumber)
            temp = temp.removePrefix(lexeme)
            typeOfPreviousElem = TokenType.HexNumber
        }

        //Выделение идентификатора
        if (Regex("""^[A-z_]""").containsMatchIn(temp)) {
            if (typeOfPreviousElem == TokenType.Identifier ||
                typeOfPreviousElem == TokenType.HexNumber
```

```
                    ) {
                        throw Exception("Expecting an element: " +
                                text.indexOf(temp))
                    }
                    lexeme = selectIdentifier(temp)
                    result.add(lexeme, TokenType.Identifier)
                    temp = temp.removePrefix(lexeme)
                    typeOfPreviousElem = TokenType.Identifier
                }

                //Выделение оператора
                if (Regex("""^[+\-*/]""").containsMatchIn(temp)) {
                    if (typeOfPreviousElem == TokenType.Operations ||
                        typeOfPreviousElem == TokenType.Assignment
                    ) {
                        throw Exception("Expecting an element: " +
                                text.indexOf(temp))
                    }
                    lexeme = selectOperator(temp)
                    result.add(lexeme, TokenType.Operations)
                    temp = temp.removePrefix(lexeme)
                    typeOfPreviousElem = TokenType.Operations
                }

                //Выделение знака присвоения
                if (temp.startsWith("=")) {
                    if (typeOfPreviousElem == TokenType.Assignment ||
                        typeOfPreviousElem == TokenType.HexNumber
                    ) {
                        throw Exception("Expecting an element: " +
                                text.indexOf(temp))
                    }
                    lexeme = selectAssigment(temp)
                    result.add(lexeme, TokenType.Assignment)
                    temp = temp.removePrefix(lexeme)
                    typeOfPreviousElem = TokenType.Assignment
                }

                //Выделение терминального знака
                if (temp.startsWith(";")) {
                    if (typeOfPreviousElem == TokenType.Terminator) {
                        throw Exception("Expecting an element: " +
                           text.indexOf(temp))
                    }
                    lexeme = temp[0].toString()
                    temp = temp.drop(1)
                    result.add(lexeme, TokenType.Terminator)
                    typeOfPreviousElem = TokenType.Terminator
                }
            }

        }
        return result
    }
}
```

TokenType.kt

```
package Work1

enum class TokenType {
    Identifier,
    HexNumber,
```

```
    Operations,
    Assignment,
    Separator,
    Terminator,
}
```

Main.kt

```
package Work1

import java.io.File

fun main(args: Array<String>) {
    //Входные данны читаются из файла
    //имя файла передается аругментом командной строки
    val inputText = readFileDirectlyAsText(args.first())


    //'Символы' доступные в 16-тиричной СС
    val digits = listOf(
        '0', 'x', '1', '2', '3', '4', '5', '6', '7', '8', '9',
        'A', 'B', 'C', 'D', 'E', 'F'
    )

    //операторы, оепредленные во входном языке
    val operators = listOf(
        '+', '-', '*', '/'
    )

    val separators = listOf(
        '{', '}', '(', ')'
    )

    //Инциализация лексического анализатора
    val lexicalAnalyzer = LexicalAnalyzer(
        elements = digits,
        operators = operators,
        separators = separators,
    )

    lexicalAnalyzer.analyze(inputText).print()
}

/**
 * Функция чтения файла
 * @param fileName имя файла, который необходимо прочитать
 * @return весь текст файла в виде одной строки
 */
fun readFileDirectlyAsText(fileName: String): String
    = File(fileName).readText(Charsets.UTF_8)
```

Результат работы программы

Входной файл:

```
red = 0xFF0000;
red = 0xFF0000;

// A long time ago in a galaxy far, far away....
{
    orange = 0xFFA500;
    red = 0xFF0000;
    red = 0xFF0000;
    red = 0xFF0000;
    red = 0xFF0000;
    {
        yellow = 0xFFFF00;
        green = 0x008000;

        red = 0xFF0000;
        red = 0xFF0000;

        {
            APIPA = 0x164256
        }
    }

    blue = 0x0000FF;

    {
        red = 0xFF0000;
        red = 0xFF0000;
        red = 0xFF0000;
        navyBlue = 0x000080;
    }
}
{
    red = 0xFF0000;
    red = 0xFF0000;
    red = 0xFF0000;
    navyBlue = 0x000080;
}
violet = 0xEE82EE;
blue = 0x0000FF;
```

Результат:

Лексема	Тип
val1	Identifier
=	Assignment
0xFFFF	HexNumber
;	Terminator
val2	Identifier
=	Assignment
val1	Identifier
+	Operations
0xAAAA	HexNumber
+	Operations
(	Separator
-	Operations
(	Separator
0x12345	HexNumber
)	Separator
)	Separator
*	Operations
var2	Identifier
;	Terminator