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 №3

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

Доработать программу лексического анализа, которая будет выполнять синтаксический разбор по заданной грамматике с построением дерева разбора. Допустимые лексемы использовать из 1 лр.

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.

Алгоритм «сдвиг - свертка»

Для использования алгоритма «сдвиг - свертка» необходимо составить отставную грамматику (грамматику с одним состоянием E) и матрицу предшествования.

Для составления матрицы предшествования необходимо найти множества крайних левых (L) и крайних правых (R) символов, а за тем при помощи составить множества крайних левых (L_t) и крайних правых терминалов (R_t).

Таблица 1. – Множества крайних левых и крайних правых символов

Символ U	**L(U)**	**R(U)**
S	a	;
F	F, T, E, (, -, a	T, E,), a
T	T, E, (, -, a	E,), a
E	(, -, a	), a

Таблица 2. – Множества крайних левых и крайних правых терминалов

Символ U	**Lt(U)**	**Rt(U)**
S	a	;
F	+, (, -, a	+,), a
T	*, /, (, -, a	*, /,), a
E	(, -, a	), a

Таблица 3. – Матрица предшествования

	;	=	a	+	-	*	/	(	)	$ (конец)
;										.>
=			.=	<.	<.			<.		
a	.>	.=		<.	<.			<.		
+			.>	.>	<.	<.	<.	<.	.>	
-			.>	<.	<.			<.		
*			>.		<.	.>	.>	<.	.>	
/			.>		<.	.>	.>	<.	.>	
(			<.	<.	.>			<.	.=	
)			.>	.>				.=	<.	
@ (начало)			<.							

Отставная грамматика P':

$$E \rightarrow a = E$$
$$E \rightarrow E + E|E$$
$$E \rightarrow E * E|E \,/\, E|E$$
$$E \rightarrow (E)|-(E)|a$$

Конфигурация автомата состоит из 3 массивов:

- Оставшаяся цепочка
- Стек
- Список инструкций свертки

Алгоритм «сдвиг-свертка» заключается переборе входной цепочки символов и выполнении действий с ними, в зависимости от отношения с символом из стека.

Если отношение – «<.» (предшествует) или «=.» (составляет основу), то символ заносится в стек (сдвиг).

Если отношение – «.>» (следует), то из стека забираются все терминалы, связанные отношением «=.» (составляет основу), и стоящие рядом с ними нетерминальные символы (E). Получившаяся цепочка ищется в правилах грамматики P’. Если найдена – в стек кладется нетерминал E, а в список инструкций добавляется номер инструкции, по которой произведена свертка.

Таким образом после того, как все выражение свернулось в одну E, по получившемуся списку инструкций можно построить дерево разбора.

Практическая часть

Язык программирования: Kotlin

SyntaxAnalyzer.kt

Класс синтаксического анализатора.

```
package Work3

data class SyntaxAnalyzer(
    val grammar: List<String>,
    val precedentMatrix: List<Pair<Char, Map<Char, TokenRelations>>>
)
{
    fun analyze(lexemeTable: LexemeTable) {
        val expression: MutableList<String> = mutableListOf()
        var memory: String
        val res = mutableListOf <Int>()

        var temp = ""
        replaceNonTerminal(lexemeTable).forEach {
            if (it != ";") temp += it
            else {
                temp += ";"
                expression.add(temp)
                temp = ""
            }
        }

        expression.forEach {
            println()
            var string = it
            var tempChar: Char
            var inGram = true
            memory = "@"

            println()
            println("$string|${memory.drop(1)}")
            while (string.isNotBlank()) {
                tempChar = string.first()

                if (tempChar !in precedentMatrix.map { pair -> pair.first })
{
                    throw Exception("unexpected character $tempChar in: $it")
                }

                memory += tempChar
                for (pair in precedentMatrix) {
                    //находим строку равную последнему символу в памяти
                    if (memory[memory.length - 2] == pair.first) {
                        for (entry in pair.second.entries) {

                            //в данной строке находим столбец равный символу,
                            //на котором сейчас головка
                            if (tempChar == entry.key) {

                                //если встречено отношение "следует", то
                                //пытаемся свернуть
                                if (entry.value == TokenRelations.SHOULD) {
                                    inGram =false
                                    for (str in grammar) {
                                        if (str in memory && memory.count{ch
```

```
                                                -> ch == 'a'} > 1) {
println("$string|${memory.drop(1).dropLast(1)}\t СВЕРТКА,
${grammar.indexOf(str) + 1}")
                                                res.add(grammar.indexOf(str))
                                                memory =
memory.reversed().replaceFirst(str, "E").reversed()
                                                inGram = true
                                                break
                                            }
                                        }
                                    }
                                    if (!inGram ||
                                        memory.contains("-a") ||
                                        memory.contains("-E")) throw
Exception("Unexpected character in: $it")
                                }
                            }
                        }
                    }
                    string = string.drop(1)
                    println("$string|${memory.drop(1)}")
                }

                while (memory != "@E") {
                    println("$string|${memory.drop(1)}")
                    for (str in grammar) {
                        if (str in memory) {
                            println("$string|${memory.drop(1)}" + "\t СВЕРТКА,
${grammar.indexOf(str) + 1}")
                            res.add(grammar.indexOf(str))
                            memory = if (str == "a" && "a=" in memory) {
                                memory.reversed().replaceFirst(str,
"E").reversed()
                            } else memory.replaceFirst(str, "E")
                            println("$string|${memory.drop(1)}")
                            break
                        }
                    }
                }
                println()
                println()
                println("Дерево синтаксического разбора выражения: $it")
                println("E")
                printThree(0, res.reversed())
            }


        }


        fun printThree(level: Int, res: List<Int>) {
            for (ch in grammar[res[level]]) {
                println(" ".repeat(level + 1) + ch)
                if (ch == 'E') {
                    if (level != res.size)  printThree(level + 1, res)
                }
            }


        }
```

```
    private fun replaceNonTerminal(lexemeTable: LexemeTable): List<String> {

        for (i in lexemeTable.content.indices) {
            if (lexemeTable.content[i].second == TokenType.HexNumber ||
                lexemeTable.content[i].second == TokenType.Identifier) {
                lexemeTable.content[i] = Pair("a", TokenType.HexNumber)
            }
        }

        return lexemeTable.content.map { it.first }
    }
}
```

TokenRelationship.kt

Перечисление отношений для заполнения матрицы предшествования

```
package Work3

enum class TokenRelations(val mean: Char) {
    BASIC('='),
    PRECEDED('<'),
    SHOULD('>')
}
```

TokenType.kt

```
package Work1

enum class TokenType {
    Identifier,
    HexNumber,
    Operations,
    Assignment,
    Separator,
    Terminator,
}
```

Entry.kt

Класс – запись в таблице

```
package Work2

/**
 * Запись в таблице идентификаторов
 * @param identifier сам идентификатор
 * @param level область видимости
 */
data class Entry(
    val identifier: String,
    val level: String
)
```

LexicalAnalyzer.kt

```
package Work2

import Work2.Tables.Table

/**
 * Лексический анализатор
 */
data class LexicalAnalyzer(
    val elements: List<Char>,
    val operators: List<Char>,
```

```
    val separators: List<Char>,
    val table: Table
) {
    /**
     * 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 шестнадцатиричного 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))
                        }
                        table.initializeScope()
                    }
                    if (lexeme == "}") table.finalizeScope()

                    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)

                    table.insert(lexeme)

                    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
    }
}
```

Main.kt

```
package Work3

import java.io.File
import java.io.PrintWriter

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

    //'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(
        '+', '-', '*', '/'
    )

    //разделители
    val separators = listOf(
        '(', ')'
    )

    val precedentMatrix: List<Pair<Char, Map<Char, TokenRelations>>> =
        listOf(
            Pair(';', hashMapOf('$' to TokenRelations.SHOULD)),
            Pair('=', hashMapOf('a' to TokenRelations.BASIC, '+' to
TokenRelations.PRECEDED, '-' to TokenRelations.PRECEDED, '(' to
TokenRelations.PRECEDED)),
            Pair('a', hashMapOf(';' to TokenRelations.SHOULD, '=' to
TokenRelations.BASIC, '+' to TokenRelations.SHOULD, '-' to
TokenRelations.PRECEDED, ')' to TokenRelations.SHOULD)),
            Pair('+', hashMapOf('a' to TokenRelations.PRECEDED, '+' to
TokenRelations.SHOULD, '-' to TokenRelations.PRECEDED, '*' to
TokenRelations.PRECEDED, '/' to TokenRelations.PRECEDED, '(' to
TokenRelations.PRECEDED, ')' to TokenRelations.SHOULD)),
            Pair('-', hashMapOf('a' to TokenRelations.SHOULD, '(' to
TokenRelations.BASIC)),
            Pair('*', hashMapOf('a' to TokenRelations.PRECEDED, '-' to
TokenRelations.PRECEDED, '*' to TokenRelations.SHOULD, '/' to
TokenRelations.SHOULD, '(' to TokenRelations.PRECEDED, ')' to
TokenRelations.SHOULD)),
            Pair('/', hashMapOf('a' to TokenRelations.PRECEDED, '-' to
TokenRelations.PRECEDED, '*' to TokenRelations.SHOULD, '/' to
TokenRelations.SHOULD, '(' to TokenRelations.PRECEDED, ')' to
TokenRelations.SHOULD)),
            Pair('(', hashMapOf('a' to TokenRelations.PRECEDED, '+' to
TokenRelations.PRECEDED, '-' to TokenRelations.BASIC, '(' to
TokenRelations.PRECEDED, ')' to TokenRelations.BASIC)),
            Pair(')', hashMapOf('a' to TokenRelations.SHOULD, '+' to
TokenRelations.PRECEDED, '(' to TokenRelations.BASIC, ')' to
TokenRelations.PRECEDED)),
            Pair('@', hashMapOf('a' to TokenRelations.PRECEDED))
        )

    val grammar = listOf(
```

```
            "a=E;",
            "E+E",
            "E*E",
            "E/E",
            "-(E)",
            "(E)",
            "a",
        )

        println("Грамматика")
        for (i in grammar.indices) {
            println("${i+1}) E → ${grammar[i]}")
        }


        //Инциализация лексического анализатора
        val lexicalAnalyzerAddressTable = LexicalAnalyzer(
            elements = digits,
            operators = operators,
            separators = separators,
            table = AddressTable()
        )

        val syntaxAnalyzer = SyntaxAnalyzer(
            grammar = grammar,
            precedentMatrix = precedentMatrix
        )

        val lexemeTable = lexicalAnalyzerAddressTable.analyze(inputText)
        lexemeTable.write(addressFilePrinter)

        addressFilePrinter.print("\n\n\n")
        lexicalAnalyzerAddressTable.table.write(addressFilePrinter)

        syntaxAnalyzer.analyze(lexemeTable)


        addressFilePrinter.close()
    }

    /**
     * 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:

```
val1 = 0xFFFF;
val2 = val1 +
    0xAAAA +
        (-(0x12345))
            * var2;
```

Результат:

```
Грамматика
1) E → a=E;
2) E → E+E
3) E → E*E
4) E → E/E
5) E → -(E)
6) E → (E)
7) E → a


a=a;|
=a;|a
a;|a=
;|a=a
;|a=a—— СВЕРТКА, 7
|a=E;
|a=E;
|a=E;—— СВЕРТКА, 1
|E


Дерево синтаксического разбора выражения: a=a;
E
 a
 =
 E
  a
 ;
```

```
a=a+a+(-(a))*a;|
=a+a+(-(a))*a;|a
a+a+(-(a))*a;|a=
+a+(-(a))*a;|a=a
+a+(-(a))*a;|a=a—— CBEPTKA, 7
a+(-(a))*a;|a=E+
+(-(a))*a;|a=E+a
+(-(a))*a;|a=E+a—— CBEPTKA, 7
(-(a))*a;|a=E+E+
-(a))*a;|a=E+E+(
(a))*a;|a=E+E+(-
a))*a;|a=E+E+(-(
))*a;|a=E+E+(-(a
))*a;|a=E+E+(-(a—— CBEPTKA, 2
)*a;|a=E+(-(a)
*a;|a=E+(-(a))
a;|a=E+(-(a))*
;|a=E+(-(a))*a
;|a=E+(-(a))*a— CBEPTKA, 7
|a=E+(-(a))*E;
|a=E+(-(a))*E;
|a=E+(-(a))*E;— CBEPTKA, 7
|a=E+(-(E))*E;
|a=E+(-(E))*E;
|a=E+(-(E))*E;— CBEPTKA, 5
|a=E+(E)*E;
|a=E+(E)*E;
|a=E+(E)*E;- CBEPTKA, 6
|a=E+E*E;
|a=E+E*E;
|a=E+E*E;—— CBEPTKA, 2
|a=E*E;
|a=E*E;
|a=E*E;- CBEPTKA, 3
|a=E;
|a=E;
|a=E;—— CBEPTKA, 1
|E
```

```
Дерево синтаксического разбора выражения: a=a+a+(-(a))*a;
E
 a
 =
 E
  E
   E
    (
    E
     -
     (
     E
      a
     )
    )
   +
   E
    (
    E
     -
     (
     E
      a
     )
    )
  *
  E
   E
    (
    E
     -
     (
     E
      a
     )
    )
   +
   E
    (
    E
     -
     (
     E
```

```
        a
      )
    )
;
```