Государственное образовательное учреждение

высшего профессионального образования

«Уфимский Государственный Авиационный Технический Университет»

Кафедра «Вычислительной математики и кибернетики»

Отчет

По лабораторной работе

по дисциплине: Методы построения трансляторов

Лабораторная работа № 2.

Проектирование лексического анализатора

Вариант 3

Проверил

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Выполнил:

студент гр. ПРО-301в

Доронин С.Г.

Уфа 2014

# Задание

Написать программу, которая выполняет лексический анализ входного теста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличии во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами. Программа должна допускать наличие комментариев неограниченной длины во входном файле. Форму организации комментариев предлагается выбрать самостоятельно.

Входной язык содержит операторы условия типа if … then … else и if … then разделенные символом ; (точка с запятой). Операторы условия содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой, знак присваивания (:=).

# Описание КС-грамматики входного языка в форме Бэкуса-Наура

<комментарий {}> **::=** {<любое количество любых символов кроме }>}

<оператор присваивания :=> **::=** := <идентификатор> | <константа>

<условный оператор if> **::=** if <булево выражение> then <оператор> [else <оператор>]

<целое число> ::= <любое количество символов от 0 до 9>

<вещественное число> **::=** <символ от 0 до 9>.< любое количество символов от 0 до 9>

<идентификатор> **::=** <любой символ от a-z> | <любой символ от A-Z> | \_<любой число-буквенный символ>

<операторы сравнения> **::=** < | <= | > | >= | ==

# Описание алгоритма работы сканера для распознавания цепочек

Текущее состояние устанавливается в начальное.

Входной файл разбивается на строки, выполняются два цикла, первый по строкам, второй по символам строки.

Далее происходит выбор дальнейших действий в зависимости от текущего состояния.

currentState **match** {

**case** H **|** F **=>** hState(char)

**case** C **=>** cState(char)

**case** G **=>** gState(char)

**case** V **=>** vState(char)

**case** D **=>** dState(char)

**case** E **=>** eState(char)

**case** L **=>** lState(char)

**case** GR **=>** grState(char)

**case** P **=>** pState(char)

}

Состояние H – начальное состояние.

Состояние F – конечное состояние.

Состояние C – обработка комментария.

Состояние G - обработка знака присваивания

Состояние V - обработка идентификатора

Состояние D - обработка числовых констант

Состояние E – обработка оператора сравнения (==)

Состояние L - обработка оператора сравнения (< <=)

Состояние GR - обработка оператора сравнения (> >=)

Состояние P - обработка вещественных чисел

## Начальное состояние

Если символ { - то поменять текущее состояние на C.

Если символ : - то поменять текущее состояние на G.

Если символ = - то поменять текущее состояние на E.

Если символ < - то поменять текущее состояние на L.

Если символ > - то поменять текущее состояние на GR.

Если символ ( или ) или ; - то добавить текущий символ в таблицу лексем и поменять состояние на F.

Если незначащий символ (пробел, знак табуляции), то поменять состояние на F.

Если символ буква, то поменять текущее состояние на V и присвоить имени идентификатора текущий символ.

Если символ цифра, то поменять текущее состояние на D и присвоить числовой переменной текущий символ.

/\*\* Обработка начального состояния \*/

**private** **def** hState(char: String) {

**char** **match** {

// Если начало комментария

**case** "{" **=>** changeCurrentState(AutoPos.C)

// Если начало знака присваивания

**case** ":" **=>** changeCurrentState(AutoPos.G)

// Если оператор сравнения

**case** "=" **=>** changeCurrentState(AutoPos.E)

**case** "<" **=>** changeCurrentState(AutoPos.L)

**case** ">" **=>** changeCurrentState(AutoPos.GR)

// Если символ-разделитель

**case** "(" **|** ")" **|** ";" **=>**

addKeyToList(char)

changeCurrentState(AutoPos.F)

// Если незначащий символ

**case** \_ **=>** **if** (isWhitespace(char)) {

changeCurrentState(AutoPos.F)

} **else** **if** (isLetter(char)) {

changeCurrentState(AutoPos.V)

currentIdName **=** **char**

} **else** **if** (isDigit(char)) {

// Если цифра

changeCurrentState(AutoPos.D)

currentNumberConst **=** **char**

} **else** {

// Что-то еще

notSupportCase(char)

}

}

}

Таким образом продолжается обработка всех остальных состояний.

## Обработка вещественных чисел

Если символ { - то добавить текущую числовую переменную в таблицу лексем и поменять текущее состояние на C.

Если символ : - то добавить текущую числовую переменную в таблицу лексем и поменять текущее состояние на G.

Если символ цифра, добавить к числовой переменной текущий символ.

Если незначащий символ (пробел, знак табуляции), то добавить текущую числовую переменную в таблицу лексем и поменять состояние на F.

Если символ ( или ) или ; - то добавить текущую числовую переменную в таблицу лексем и поменять состояние на F.

Если незначащий символ (пробел, знак табуляции), то добавить текущую числовую переменную в таблицу лексем и поменять состояние на F.

/\*\* Обработка вещественных чисел \*/

**private** **def** pState(char: String) {

**val** isReal **=** **true**

**char** **match** {

// Если начало комментария

**case** "{" **=>**

addConstToList(currentNumberConst, isReal)

changeCurrentState(AutoPos.C)

// Если начало знака присваивания

**case** ":" **=>**

addConstToList(currentNumberConst, isReal)

changeCurrentState(AutoPos.G)

// Если цифры

**case** \_ **=>** **if** (isDigit(char)) {

currentNumberConst **+=** **char**

logger.debug(s"\t new digit: $currentNumberConst")

} **else** **if** (isWhitespace(char)) {

// Если незначащий символ

addConstToList(currentNumberConst, isReal)

changeCurrentState(AutoPos.F)

} **else** **if** (isListened(char, "();")) {

// Если символ-разделитель

addConstKeyToList(currentNumberConst, char, isReal)

changeCurrentState(AutoPos.F)

} **else** {

// Что-то еще

notSupportCase(char)

}

}

}

# Текст программы

## Конечный автомат

**object** LexAuto {

*/\*\* Синтаксический анализ выполнен без ошибок \*/*

**val** NoErrors = -1

}

*/\*\**

*\* Класс - моделирующий работу КА, на основе которого построен лексический распознаватель*

*\*/*

**class** LexAuto **extends** ElementAdder {

*/\*\* Дополнитель конца строки \*/*

**private val** LineEnd = **" "**

*////////////////////////////////////////////////////////*

*/\*\* Текущее состояние автомата \*/*

**var** currentState = AutoPos.H

*/\*\* Предыдущее состояние автомата (для отладки) \*/*

**var** prevState = AutoPos.H

*/\*\* Логирование \*/*

**private val** logger = Logger(infoEnable = **true**, debugEnable = **true**, traceEnable = **true**)

*/\*\* Текущее имя идентификатора \*/*

**private var** currentIdName = **""**

*/\*\* Текущая числовая константа \*/*

**private var** currentNumberConst = **""**

*////////////////////////////////////////////////////////*

*/\*\* Инициализация \*/*

**private def** init(listLex: ListBuffer[LexElem]) {

**this**.lexList = listLex

reset()

}

*/\*\**

*\* Моделирование работы КА*

*\** **@param lines** *Текст программы*

*\** **@param listLex** *Таблица найденных лексем*

*\** **@return** *0 если лексический анализ выполнен без ошибок,*

*\* а если ошибка обнаружена - номер строки в исходном файле,*

*\* в которой она присутствует*

*\*/*

**def** makeLexList(lines: Array[String], listLex: ListBuffer[LexElem]): Int = {

**var** char = **""**

**try** {

*// Инициализация*

init(listLex)

lines.foreach { l =>

columnIndex = 0

*// Обрабатываемая строка*

**val** line = lines(lineIndex) + LineEnd

line.foreach { c =>

*// Обрабатываемый символ*

char = c.toString

logger.debug(**s"char: '$**char**'"**)

**import** mpt.lab.two.automat.AutoPos.\_

currentState **match** {

**case** H | F => hState(char)

**case** C => cState(char)

**case** G => gState(char)

**case** V => vState(char)

**case** D => dState(char)

**case** E => eState(char)

**case** L => lState(char)

**case** GR => grState(char)

**case** P => pState(char)

}

columnIndex += 1

fromBeginIndex += 1

}

lineIndex += 1

}

LexAuto.NoErrors

} **catch** {

**case** e: MatchError =>

addInfoToList(**s"Ошибка при разборе '$**char**'"**)

currentPosition.fromBegin

}

}

*/\*\* Сброс состояния автомата \*/*

**private def** reset() {

lineIndex = 0

columnIndex = 0

fromBeginIndex = 0

currentState = AutoPos.H

}

*/\*\* Обработка начального состояния \*/*

**private def** hState(char: String) {

char **match** {

*// Если начало комментария*

**case "{"** => changeCurrentState(AutoPos.C)

*// Если начало знака присваивания*

**case ":"** => changeCurrentState(AutoPos.G)

*// Если оператор сравнения*

**case "="** => changeCurrentState(AutoPos.E)

**case "<"** => changeCurrentState(AutoPos.L)

**case ">"** => changeCurrentState(AutoPos.GR)

*// Если символ-разделитель*

**case "("** | **")"** | **";"** =>

addKeyToList(char)

changeCurrentState(AutoPos.F)

*// Если незначащий символ*

**case** \_ => **if** (isWhitespace(char)) {

changeCurrentState(AutoPos.F)

} **else if** (isLetter(char)) {

changeCurrentState(AutoPos.V)

currentIdName = char

} **else if** (isDigit(char)) {

*// Если цифра*

changeCurrentState(AutoPos.D)

currentNumberConst = char

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Обработка комментария \*/*

**private def** cState(char: String) {

char **match** {

*// Конец комментария*

**case "}"** => changeCurrentState(AutoPos.F)

*// Текст комментария*

**case** \_ => **if** (isAnyChar(char, **"}"**)) {

logger.debug(**s"Comment char: $**char**"**)

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Обработка знака присваивания \*/*

**private def** gState(char: String) {

char **match** {

*// Конец знака присваивания*

**case "="** =>

addOperatorToList(LexOperators.Assignment)

changeCurrentState(AutoPos.F)

*// Что-то еще*

**case** \_ => notSupportCase(char)

}

}

*/\*\* Обработка идентификатора \*/*

**private def** vState(char: String) {

char **match** {

*// Если начало комментария*

**case "{"** =>

addWordToList(currentIdName)

changeCurrentState(AutoPos.C)

*// Если начало знака присваивания*

**case ":"** =>

addWordToList(currentIdName)

changeCurrentState(AutoPos.G)

*// Если буквы и цифры*

**case** \_ => **if** (isLetter(char) || isDigit(char)) {

currentIdName += char

logger.debug(**s"\t new name: $**currentIdName**"**)

} **else if** (isWhitespace(char)) {

*// Если незначащий символ*

addWordToList(currentIdName)

changeCurrentState(AutoPos.F)

} **else if** (isListened(char, **"().;"**)) {

*// Если символ-разделитель*

addWordKeyToList(currentIdName, char)

changeCurrentState(AutoPos.F)

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Обработка числовых констант \*/*

**private def** dState(char: String) {

char **match** {

*// Если начало комментария*

**case "{"** =>

addConstToList(currentNumberConst)

changeCurrentState(AutoPos.C)

*// Если начало знака присваивания*

**case ":"** =>

addConstToList(currentNumberConst)

changeCurrentState(AutoPos.G)

*// Если разделитель вещественного числа*

**case "."** =>

changeCurrentState(AutoPos.P)

currentNumberConst += char

*// Если цифры*

**case** \_ => **if** (isDigit(char)) {

currentNumberConst += char

logger.debug(**s"\t new digit: $**currentNumberConst**"**)

} **else if** (isWhitespace(char)) {

*// Если незначащий символ*

addConstToList(currentNumberConst)

changeCurrentState(AutoPos.F)

} **else if** (isListened(char, **"();"**)) {

*// Если символ-разделитель*

addConstKeyToList(currentNumberConst, char)

changeCurrentState(AutoPos.F)

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Обработка вещественных чисел \*/*

**private def** pState(char: String) {

**val** isReal = **true**

char **match** {

*// Если начало комментария*

**case "{"** =>

addConstToList(currentNumberConst, isReal)

changeCurrentState(AutoPos.C)

*// Если начало знака присваивания*

**case ":"** =>

addConstToList(currentNumberConst, isReal)

changeCurrentState(AutoPos.G)

*// Если цифры*

**case** \_ => **if** (isDigit(char)) {

currentNumberConst += char

logger.debug(**s"\t new digit: $**currentNumberConst**"**)

} **else if** (isWhitespace(char)) {

*// Если незначащий символ*

addConstToList(currentNumberConst, isReal)

changeCurrentState(AutoPos.F)

} **else if** (isListened(char, **"();"**)) {

*// Если символ-разделитель*

addConstKeyToList(currentNumberConst, char, isReal)

changeCurrentState(AutoPos.F)

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Обработка оператора сравнения (==) \*/*

**private def** eState(char: String) {

char **match** {

*// Конец оператора сравнения (==)*

**case "="** =>

addOperatorToList(LexOperators.Equals)

changeCurrentState(AutoPos.F)

*// Что-то еще*

**case** \_ => notSupportCase(char)

}

}

*/\*\* Обработка оператора сравнения (< <=) \*/*

**private def** lState(char: String) {

char **match** {

*// Конец оператора сравнения (<=)*

**case "="** =>

addOperatorToList(LexOperators.LE)

changeCurrentState(AutoPos.F)

*// Если незначащий символ*

**case** \_ => **if** (isWhitespace(char)) {

addOperatorToList(LexOperators.L)

changeCurrentState(AutoPos.F)

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Обработка оператора сравнения (> >=) \*/*

**private def** grState(char: String) {

char **match** {

*// Конец оператора сравнения (>=)*

**case "="** =>

addOperatorToList(LexOperators.GE)

changeCurrentState(AutoPos.F)

*// Если незначащий символ*

**case** \_ => **if** (isWhitespace(char)) {

addOperatorToList(LexOperators.G)

changeCurrentState(AutoPos.F)

} **else** {

*// Что-то еще*

notSupportCase(char)

}

}

}

*/\*\* Не поддерживаемая операция \*/*

**private def** notSupportCase(char: String) {

logger.debug(**s"\t not support: $**char**"**)

**throw new** MatchError(**s"Not support case: '$**char**'"**)

}

*/\*\* Меняет текущее состояние автомата \*/*

**private def** changeCurrentState(newState: AutoPos) {

prevState = currentState

currentState = newState

logger.debug(**s"\t change state: $**newState**"**)

}

}

## Обработчики кнопок формы

*/\*\* Диалог выбора файла \*/*

**private val** fileChooser = **new** FileChooser(**new** File(**"data"**))

reactions += {

*// Выбор файла*

**case** ButtonClicked(`browseFileButton`) =>

**val** result = fileChooser.showDialog(**null**, **"Select file"**)

**if** (result == FileChooser.Result.Approve) {

fileContentTextArea.text = Source.fromFile(fileChooser.selectedFile).mkString

filePathTextField.text = fileChooser.selectedFile.getAbsolutePath

}

*// Обработка текста программы*

**case** ButtonClicked(`processingButton`) => processing()

*// Выход из программы*

**case** ButtonClicked(`exitButton`) => System.exit(0)

}

*/\*\* Обработка текста программы \*/*

**private def** processing() {

*// Очистка таблицы*

lexemModel.clear()

*// Разбор текста*

**val** auto = **new** LexAuto

**val** lines = fileContentTextArea.text.split(**"\n"**)

**val** out = ListBuffer[LexElem]()

**val** statusOrPos = auto.makeLexList(lines, out)

*// Установка статуса разбора текста*

parsingStatusLabel.text = **if** (statusOrPos == LexAuto.NoErrors) {

**"Разбор выполнен без ошибок"**

} **else "В ходе разбора обнаружена ошибка"**

*// Добавление лексем в таблицу*

out.zipWithIndex.map {

**case** (e, index) => Seq(**s"$**{index+1}**"**, e.lexInfo.name, e.value)

}.foreach(lexemModel.addRow)

*// Переключение на вкладку с таблицей*

tabbedPane.setSelectedIndex(1)

}

# Выводы по проделанной работе

В результате лабораторной работы №2 построен лексический анализатор на основе конечного автомата. Построенный лексический анализатор позволяет выделять в тексте исходной программы, лексемы следующих типов:

* Ключевые слова (if, then, else);
* Идентификаторы (при этом в именах идентификаторов различаются строчные и прописные английские буквы);
* Знак операции присваивания;
* Целые и вещественные десятичные константы без знака;
* Разделители (круглые скобки и точка с запятой);
* Операторы сравнения (< <= > >= ==)

Лексический анализатор игнорирует в тексте входной программы пробелы, знаки табуляции и переводы строи, а также комментарии, выделенные фигурными скобками.

В случае обнаружения неверной лексемы (например числа, содержащего букву), незакрытого комментария или незавершенной лексемы, лексический анализатор выдает сообщение об ошибке и прекращает дальнейший анализ. При наличии нескольких неверных лексем анализатор обнаруживает только первую из них.

Результатом выполнения лексического анализатора является структура данных, которая представляет таблицу лексем.