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 № 3.

Построение простейшего дерева вывода

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, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике с построением дерева разбора. Текст на входном языке задается в виде символьного (текстового) файла.

Допускается исходить из условия, что текст содержит не более одного предложения входного языка.

При наличии во входном файле текста, соответствующего заданному языку, программа должна строить и отображать дерево синтаксического разбора. Если же текст во входном файле содержит ошибки (лексические или синтаксические), программа должна выдавать сообщение о наличии ошибок во входном тексте и корректно завершать свое выполнение.

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 (:=).

# Цель работы

Изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования.

# Заданная грамматика входного языка

E **->** E; **-** правило 1;

E **->** **if** E then E **else** E **|** **if** E then E **|** a :**=** E **-** правила 2, 3, и 4;

E **->** **if** E then E **else** E **|** **if** a :**=** E **-** правила 5 и 6;

E **->** E **<** E **|** E **>** E **|** E **-** правила 7, 8, 9;

E **->** E **==** E **|** E **-** правила 10 и 11;

E **->** a **|** (E) **-** правила 12 и 13

# Множества крайних правых и крайних левых символов

Символ U **|** L(U) **|** R(U)

**---------------------------------------------------**

S **|** F, if, a **|** ;

F **|** if, a **|** F, E, D, C, a, )

T **|** if, a **|** T, E, D, C, a, )

E **|** E, D, C, a, ( **|** D, C, a, )

D **|** D, C, a, ( **|** C, a, )

C **|** a, ( **|** a, )

# Множества крайних правых и крайних левых терминальных символов

Символ U **|** L(U) **|** R(U)

**---------------------------------------------------**

S **|** if, a, ; **|** ;

F **|** if, a **|** else, then, :**=**, **<**, **>**, **==**, a, )

T **|** if, a **|** else, :**=**, **<**, **>** **==**, a, )

E **|** **<**, **>**, **==**, a, ( **|** **<**, **>**, **==**, a, )

D **|** **==**, a, ( **|** **==**, a, )

C **|** a, ( **|** a, )

# Матрица предшествования для грамматики

**val** GrammMatrix: Array[Array[Char]] **=** Array(

Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>'), // ;

Array(' ', ' ', '=', ' ', '<', ' ', '<', '<', '<', '<', ' ', ' '), // if

Array('>', '<', ' ', '=', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' '), // then

Array('>', '<', ' ', '>', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' '), // else

Array('>', ' ', '>', '>', ' ', '=', '>', '>', '>', ' ', '>', ' '), // a

Array('>', ' ', '>', '>', '<', ' ', '<', '<', '<', '<', ' ', ' '), // :=

Array('>', ' ', '>', '>', '<', ' ', '>', '>', '<', '<', '>', ' '), // <

Array('>', ' ', '>', '>', '<', ' ', '>', '>', '<', '<', '>', ' '), // >

Array('>', ' ', '>', '>', '<', ' ', '>', '>', '>', '<', '>', ' '), // ==

Array(' ', ' ', ' ', ' ', '<', ' ', '<', '<', '<', '<', '=', ' '), // (

Array('>', ' ', '>', '>', ' ', ' ', '>', '>', '>', ' ', '>', ' '), // )

Array('<', '<', ' ', ' ', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ') // Начало

)

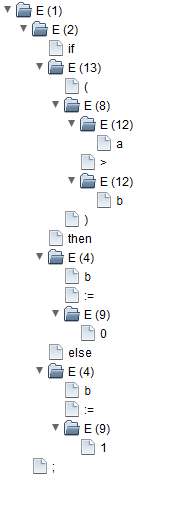
# Пример выполнения разбора

## Предложение на входном языке

{Тестирование синтаксического разбора}

**if** **(**a **>** b**)** **then** b **:=** 0 **else** b **:=** 1**;**

## Синтаксическое дерево



В скобках указаны номера правил для свертки.

# Текст программы

## Обработчики формы

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()

**val** root **=** TSymbol.createLex(LexElem.createKey("", Position(0, 0, 0)))

syntaxTree.model **=** TreeModel(root)(\_.children)

// Разбор текста

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

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

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

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

// Установка статуса разбора текста

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

lexParsingStatusLabel.text **=** "Лексический разбор выполнен без ошибок"

// Добавление конечной лексемы

**val** lexListForSyntax **=** lexList.toList :**+** LexControl.LexStop

**val** symbStack **=** **new** TSymbStack

**val** rootSymbol **=** SyntSymb.buildSyntList(lexListForSyntax, symbStack)

syntaxTree.model **=** TreeModel(rootSymbol)(\_.children)

syntaxTree.expandAll()

**if** (rootSymbol.lexem **==** null) {

syntaxParsingStatusLabel.text **=** "Синтаксический разбор выполнен без ошибок"

} **else** {

syntaxParsingStatusLabel.text **=** "В ходе синтаксического разбора обнаружена ошибка"

}

// Переключение на вкладку с синтаксическим деревом

tabbedPane.setSelectedIndex(2)

} **else** {

lexParsingStatusLabel.text **=** "В ходе лексического разбора обнаружена ошибка"

syntaxParsingStatusLabel.text **=** ""

// Переключение на вкладку с таблицей лексем

tabbedPane.setSelectedIndex(1)

}

// Добавление лексем в таблицу

lexList.zipWithIndex.map {

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

}.foreach(lexemModel.addRow)

}

## Модуль описания матрицы предшествования и правил грамматики

/\*\*

\* Модуль описания матрицы предшествования и

\* правил грамматики

\*/

**object** SyntRule {

/\*\* Состовляет основу \*/

**val** Basis **=** '='

/\*\* Предшуствует \*/

**val** Preceded **=** '<'

/\*\* Следует \*/

**val** Follow **=** '>'

/\*\* Не сопоставимы \*/

**val** Empty **=** ' '

/\*\* Матрица операторного предшествования \*/

**val** GrammMatrix: Array[Array[Char]] **=** Array(

Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '>'), // ;

Array(' ', ' ', '=', ' ', '<', ' ', '<', '<', '<', '<', ' ', ' '), // if

Array('>', '<', ' ', '=', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' '), // then

Array('>', '<', ' ', '>', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' '), // else

Array('>', ' ', '>', '>', ' ', '=', '>', '>', '>', ' ', '>', ' '), // a

Array('>', ' ', '>', '>', '<', ' ', '<', '<', '<', '<', ' ', ' '), // :=

Array('>', ' ', '>', '>', '<', ' ', '>', '>', '<', '<', '>', ' '), // <

Array('>', ' ', '>', '>', '<', ' ', '>', '>', '<', '<', '>', ' '), // >

Array('>', ' ', '>', '>', '<', ' ', '>', '>', '>', '<', '>', ' '), // ==

Array(' ', ' ', ' ', ' ', '<', ' ', '<', '<', '<', '<', '=', ' '), // (

Array('>', ' ', '>', '>', ' ', ' ', '>', '>', '>', ' ', '>', ' '), // )

Array('<', '<', ' ', ' ', '<', ' ', ' ', ' ', ' ', ' ', ' ', ' ') // Начало

)

**val** E **=** "E"

**val** A **=** "A"

/\*\* Максимальная длина правила \*/

**val** RuleLength **=** 6

/\*\* Правила грамматики \*/

**val** \_GrammRules: Array[Array[String]] **=** Array(

Array(E, ";"), // 1

Array("if", E, "then", E, "else", E), // 2

Array("if", E, "then", E), // 3

Array(A, ":=", E), // 4

Array("if", E, "then", E, "else", E), // 5

Array(A, ":=", E), // 6

Array(E, "<", E), // 7

Array(E, ">", E), // 8

Array(E), // 9

Array(E, "==", E), // 10

Array(E), // 11

Array(A), // 12

Array("(", E, ")") // 13

)

/\*\* Правила грамматики \*/

**val** GrammRules: Array[String] **=** \_GrammRules.map(\_.mkString("|"))

/\*\* Корректировка отношения \*/

**def** correctRule(cRule: Char, lexTCur: TLexem, lex: LexType, symbStack: TSymbStack): Char **=** cRule

/\*\* Наименование нетерминального символа в правилах остновной грамматики \*/

**def** makeSymbolStr(ruleNum: Int): String **=** E

}

## Модуль, обеспечивающий выполнение функции синтаксического разбора

**object** Types {

**type** TLexem **=** LexElem

}

/\*\*

\* Модуль, обеспечивающий выполнение функции синтаксического

\* разбора с помощью алгоритма "сдвиг-свертка"

\*/

**object** SyntSymb {

/\*\* Логирование \*/

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

/\*\*

\* Сдвиг-свертка <br/>

\* Результат функции: <br/>

\* - нетерминальный символ (корень синтаксического дерева), если разбор был выполнен успешно <br/>

\* - терминальный символ, ссылающийся на лексему, где была обнаружена ошибка, если разбор выполнен с ошибками <br/>

\*/

**def** buildSyntList(listLex: List[TLexem], symbStack: TSymbStack): TSymbol **=** {

symbStack.push(LexControl.LexStart)

**val** iCnt **=** listLex.size **-** 1

**var** result: TSymbol **=** **null**

/\*\* Прерывание цикла \*/

**var** break **=** **false**

**var** i **=** 0

**while** (i **<=** iCnt **&&** **!**break) {

**val** lexTCur **=** symbStack.topLexem.get

logger.debug("lexTCur: " **+** lexTCur)

**val** lexCurFromList **=** listLex(i)

logger.debug("lexCurFromList: " **+** lexCurFromList)

// Если на вершине стека начальная лексема, а текущая лексема - конечная, то разбор завершен

**if** (lexTCur **==** LexControl.LexStart **&&** lexCurFromList **==** LexControl.LexStop) {

break **=** **true**

} **else** {

// Смотрим отношение лексемы на вершине стека и текущей лексемы в строке

**var** cRule **=** SyntRule.GrammMatrix(lexTCur.index)(lexCurFromList.index)

cRule **=** SyntRule.correctRule(cRule, lexTCur, lexCurFromList.lexInfo, symbStack)

logger.debug("cRule: " **+** cRule)

cRule **match** {

**case** '<' **|** '=' **=>** // Надо выполнить сдвиг (перенос)

symbStack.push(lexCurFromList)

i **+=** 1

**case** '>' **=>** // Надо выполнить свертку

**if** (symbStack.makeTopSymb.isEmpty) {

logger.debug("Unable to perform convolution")

// Если не удалось выполнить свертку

result **=** TSymbol.createLex(lexCurFromList)

break **=** **true**

}

**case** \_ **=>** // Отношение не установлено - ошибка разбора

result **=** TSymbol.createLex(lexCurFromList)

break **=** **true**

}

}

}

**if** (result **==** null) {

// Если разбор прошел без ошибок

**if** (symbStack.count **==** 2) {

result **=** symbStack.getSymbol(1)

} **else** {

result **=** TSymbol.createLex(listLex(iCnt))

}

} **else** {

// Добавление иформации об ошибке

**val** posLex **=** LexElem.createInfo("Position", result.lexem.pos)

**val** stackStr **=** s"Stack: ${symbStack.toString}"

**val** stackLex **=** LexElem.createKey(stackStr, **new** Position(0, 0, 0))

result.children **=** List(posLex, stackLex).map(TSymbol.createLex)

}

result

}

}

## Описание всех данных, связанных с понятием «символ грамматики»

/\*\*

\* Описание всех данных, связанных с понятием "символ грамматики"

\*/

**class** TSymbol {

/\*\* Информация о символе \*/

**private** **var** symbInfo: TSymbInfo **=** **null**

/\*\*

\* Номер правила, которым создан символ

\* Для терминальных символов = 0,

\* для нетерминальных символов он может быть от 1 до 13

\*/

**private** **var** iRuleNum: Int **=** 0

/\*\* Номер правила, которым создан символ \*/

**def** ruleNum **=** iRuleNum

/\*\* Получение символа из правила по номеру символа \*/

**def** getItem(symbolIndex: Int): TSymbol **=** symbInfo.lexList(symbolIndex)

/\*\* Получение кол-ва символов в правиле \*/

**def** count: Int **=** symbInfo.lexList.size

/\*\* Формирование строкового представления символа \*/

**def** symbolStr: String **=** {

**if** (symbType **==** TSymbKind.SymbSynt) {

SyntRule.makeSymbolStr(iRuleNum)

} **else** {

lexem.value

}

}

/\*\* Тип символа \*/

**def** symbType **=** symbInfo.symbType

/\*\* Ссылка на лексему для терминального символа \*/

**def** lexem **=** symbInfo.lexOne

/\*\* Дочерние элементы \*/

**def** children **=** symbInfo.lexList

/\*\* Добавление дочерних элементов \*/

**def** children\_**=**(childrenList: List[TSymbol]) **=** symbInfo.lexList **=** childrenList

}

**object** TSymbol {

/\*\* Создание терминального символа по лексеме \*/

**def** createLex(lex: TLexem) **=** {

**new** TSymbol {

symbInfo **=** **new** TSymbInfo(TSymbKind.SymbLex, lex, List[TSymbol]())

// Правило не используется пэ 0

iRuleNum **=** 0

}

}

/\*\* Создание нетерминального символа на основе правила и массива символов \*/

**def** createSymb(iR: Int, iSymbN: Int, symbArr: ListBuffer[TSymbol]) **=** {

**new** TSymbol {

**val** lexList **=** symbArr.reverse.toList

symbInfo **=** **new** TSymbInfo(TSymbKind.SymbSynt, null, lexList)

// Запоминаем номер правила

iRuleNum **=** iR

}

}

}

/\*\*

\* Структура данных для символа грамматики

\*/

**case** **class** TSymbInfo(symbType: TSymbKind, lexOne: TLexem, **var** lexList: List[TSymbol])

/\*\*

\* Типы символов

\*/

**object** TSymbKind **extends** Enumeration {

**type** TSymbKind **=** Value

/\*\* Терминальный символ (лексема) \*/

**val** SymbLex **=** Value

/\*\* Нетерминальный символ \*/

**val** SymbSynt **=** Value

}

## Синтаксический стек

/\*\*

\* Синтаксический стек

\*/

**class** TSymbStack {

/\*\* Символы \*/

**private** **val** items **=** ListBuffer[TSymbol]()

/\*\* Логирование \*/

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

/\*\* Очистка стека \*/

**def** clear() **=** items.clear()

/\*\* Выборка символа по номеру от вершины стека \*/

**def** getSymbol(index: Int): TSymbol **=** items(index)

/\*\* Помещения в стек входящей лексемы \*/

**def** push(lex: TLexem): TSymbol **=** {

**val** lexSymb **=** TSymbol.createLex(lex)

items **+=** lexSymb

lexSymb

}

/\*\* Самая верхняя лексема в стеке \*/

**def** topLexem: Option[TLexem] **=** {

**val** find **=** items.reverse.find(\_.symbType **==** TSymbKind.SymbLex)

**if** (find.isDefined) {

Some(find.get.lexem)

} **else** None

}

/\*\* Удаление элемента из стека \*/

**def** delete(index: Int) **=** items.remove(index)

/\*\* Свертка и помещение нового символа на вершину стека \*/

**def** makeTopSymb: Option[TSymbol] **=** {

/\*\* Массив хранения символов правила \*/

**val** symbArr **=** ListBuffer[TSymbol]()

/\*\* Счетчик символов в стеке \*/

**var** i **=** 0

/\*\* Список элементов правила \*/

**var** sRuleList **=** ListBuffer[String]()

/\*\* Текущий символ \*/

**var** symCur: TSymbol **=** **null**

**val** addToRule **=** (sStr: String, sym: TSymbol) **=>** {

symCur **=** sym

symbArr **+=** getSymbol(i)

sRuleList **+=** sStr

delete(i)

}

i **=** items.size **-** 1

**var** break **=** **false**

**while** (i **>=** 0 **&&** **!**break) {

**val** s **=** getSymbol(i)

**if** (s.symbType **==** TSymbKind.SymbSynt) {

addToRule(s.symbolStr, symCur)

} **else** {

**if** (symCur **==** null) {

addToRule(s.lexem.valueWithReplace, s)

} **else** {

**val** rowIndex **=** s.lexem.index

**val** columnIndex **=** symCur.lexem.index

**if** (SyntRule.GrammMatrix(rowIndex)(columnIndex) **==** SyntRule.Basis) {

addToRule(s.lexem.valueWithReplace, s)

} **else** {

break **=** **true**

}

}

}

**if** (symbArr.size **>** SyntRule.RuleLength) break **=** **true**

i **-=** 1

}

**var** symbol: Option[TSymbol] **=** None

// Если выбран хотя бы один символ из стека

**if** (symbArr.size **>** 0) {

**val** sRuleStr **=** sRuleList.reverse.mkString("|")

logger.debug("\tsRule: " **+** sRuleStr)

**val** ruleIndex **=** SyntRule.GrammRules.indexOf(sRuleStr)

**if** (ruleIndex **!=** **-**1) {

symbol **=** Some(TSymbol.createSymb(ruleIndex**+**1, symbArr.size, symbArr))

items **+=** symbol.get

} **else** {

symbArr.clear()

}

}

symbol

}

/\*\* Кол-во элементов в стеке \*/

**def** count **=** items.size

**override** **def** toString **=** items.map(\_.symbolStr).mkString(" | ")

}