Институт информационных технологий

Кафедра: Математическое и программное обеспечение ЭВМ

Дисциплина: Теория автоматов и формальных языков

ЛАБОРАТОРНАЯ РАБОТА № 7-8

Тема: **Построение дескрипторного кода и псевдокода. Построение регулярной̆ грамматики**

Выполнил:

студент гр. 1ПИб-02-2оп-23

Кринкин Олег Алексеевич

Проверил:

доцент, к.т.н. Ганичева Оксана Георгиевна

ЗАДАНИЕ

1) Написать часть/функцию лексического анализатора, обрабатывающую входную строку по конечному автомату на основе оператора выбора,формирование таблиц с классами лексем**, (это часть с предыдущей работы ) реализующую построение дескрипторного кода и псевдокода.**

**Результат работы программы:**

**1. считываемая входная строка (код)**

**2. таблицы с выделенными классами лексем (в которых содержится информация для формирования дескрипторного и псевдокодов).**

**3. строка с сформированным дескрипторным кодом**

**4. строка с сформированным псевдокодом**

**5. В программе предусмотреть логичное формирование дескрипторного и псевдокода для повторных символов (проверить на наборах типа ++++++ и <<<<<< и т.д.)**

**Преподавателю прислать отчет в следующем виде.**

**1. код программы**

**2. скриншоты с результатами работы (сделать их читаемыми по размерам)**

**2) Построить регулярную грамматику, задающую тот же язык, что и конечный автомат (из лабораторной работы 3\_4). Описание грамматики должно содержать совмещенную таблицу соответствия команды КА и правила РГ.**

Оформить все в единый отчет 1) и 2)

ХОД РАБОТЫ

Для построения дескрипторного и псевдокода была модифицирована функция построения таблиц следующим образом:

|  |
| --- |
| System::Void Analize::GUI::addToTable(int state, String^ lexem) {  lexem = lexem->Replace("\n", "\\n")->Replace("\r", "\\r");  switch (state) {  case 1: { // Ключевое слово  buildCodes(state, lexem, lexem, dataGridViewKeys);  break;  }  case 2: { // Константа  buildCodes(state, lexem, "const", dataGridViewConst);  break;  }  case 3: { // Идентификатор  buildCodes(state, lexem, "id", dataGridViewIDs);  break;  }  case 4: { // Оператор отношения  buildCodes(state, lexem, lexem, dataGridViewRelative);  break;  }  case 5: { // Знак операции  buildCodes(state, lexem, lexem, dataGridViewSigns);  break;  }  case 6: { // Разделитель  buildCodes(state, lexem, lexem, dataGridViewDelims);  break;  }  }  } |

Теперь функция построения таблиц вызывает отдельный метод buildCodes(), который отвечает и за добавление лексем в таблицу, и построение кода:

|  |
| --- |
| System::Void Analize::GUI::buildCodes(int state, String^ lexem, String^ code, DataGridView^ gridView) {  if (!getFromTable(lexem, gridView)) {  int k\_count = gridView->RowCount++;  gridView->Rows[k\_count - 1]->Cells[0]->Value = k\_count.ToString();  gridView->Rows[k\_count - 1]->Cells[1]->Value = lexem;  gridView->Rows[k\_count - 1]->Cells[2]->Value = code;  }  textBoxDescript->Text += "(" + state \* 10 + "," + getFromTable(lexem, gridView) + ")";  textBoxPseudo->Text += code;  } |

Метод добавляет к тексту кода код каждой лексемы из таблицы. Получение кода лексемы для дескрипторного кода ищется в таблице при помощи метода buildCodes(), который производит последовательный поиск по таблице лексем и возвращает её номер.

Для конечного автомата из лабораторной работы 3-4 (по 2 классам лексем) построено его описание и эквивалентная ему регулярная грамматика. Совмещённая таблица правил автомата и грамматики представлена ниже:

|  |  |
| --- | --- |
| Формальное описание конечного автомата | Формальное описание регулярной грамматики по заданному конечному автомату |
| X = {A…Z, a…z, 0…9, +, –, .}  S = {S0, S1, S2, … S66, S67, S68}  S0 = {S0}  F = {S1, S2},  где:  S1 – ключевые слова  S2 – константы  δ: |  |
| (S0, a) → S7  (S0, b) → S10  (S0, c) → S16  (S0, d) → S24  (S0, e) → S33  (S0, f) → S36  (S0, i) → S41  (S0, p) → S43  (S0, r) → S53  (S0, t) → S58  (S0, v) → S61  (S0, -) → S65  (S0, +) → S65  (S0, 0) → S64  (S0, 1) → S64  (S0, 2) → S64  (S0, 3) → S64  (S0, 4) → S64  (S0, 5) → S64  (S0, 6) → S64  (S0, 7) → S64  (S0, 8) → S64  (S0, 9) → S64  (S0, .) → S66 | S0 → aS7 | bS10 | cS16 | dS24 | eS33 | fS36 | iS41 | pS43 | rS53 | tS58 | vS61 | -S65 | +S66 | 0S64 | 1S64 | 2S64 | 3S64 | 4S64 | 5S64 | 6S64 | 7S64 | 8S64 | 9S64 | .S66 |
| (S7, u) → S8 | S7 → uS8 |
| (S8, t) → S9 | S8 → tS9 |
| (S9, o) → S1 | S9 → o |
| (S10, r) → S11  (S10, o) → S14 | S10 → rS11 | oS14 |
| (S11, e) → S12 | S11 → eS12 |
| (S12, a) → S13 | S12 → aS13 |
| (S13, k) → S1 | S13 → k |
| (S14, o) → S15 | S14 → oS15 |
| (S15, l) → S1 | S15 → l |
| (S16, l) → S17  (S16, h) → S20  (S16, a) → S22 | S16 → lS17 | hS20 | aS22 |
| (S17, a) → S18 | S17 → aS18 |
| (S18, s) → S19 | S18 → sS19 |
| (S19, s) → S1 | S19 → s |
| (S20, a) → S21 | S20 → aS21 |
| (S21, r) → S1 | S21 → r |
| (S22, s) → S23 | S22 → sS23 |
| (S23, e) → S1 | S23 → e |
| (S24, o) → S25  (S24, e) → S29 | S24 → oS25 | eS29 |
| (S25, u) → S26 | S25 → uS26 |
| (S26, b) → S27 | S26 → bS27 |
| (S27, l) → S28 | S27 → lS28 |
| (S28, e) → S1 | S28 → e |
| (S29, l) → S30 | S29 → lS30 |
| (S30, e) → S31 | S30 → eS31 |
| (S31, t) → S32 | S31 → tS32 |
| (S32, e) → S1 | S32 → e |
| (S33, l) → S34 | S33 → lS34 |
| (S34, s) → S35 | S34 → sS35 |
| (S35, e) → S1 | S35 → e |
| (S36, a) → S37  (S36, o) → S40 | S36 → aS37 | oS40 |
| (S37, l) → S38 | S37 → lS38 |
| (S38, s) → S39 | S38 → sS39 |
| (S39, e) → S1 | S39 → e |
| (S40, r) → S1 | S40 → r |
| (S41, n) → S42  (S41, f) → S1 | S41 → nS42 | f |
| (S42, t) → S1 | S42 → t |
| (S43, r) → S44  (S43, u) → S49 | S43 → rS44 | uS49 |
| (S44, i) → S45 | S44 → iS45 |
| (S45, v) → S46 | S45 → vS46 |
| (S46, a) → S47 | S46 → aS47 |
| (S47, t) → S48 | S47 → tS48 |
| (S48, e) → S1 | S48 → e |
| (S49, b) → S50 | S49 → bS50 |
| (S50, l) → S51 | S50 → lS51 |
| (S51, i) → S52 | S51 → iS52 |
| (S52, c) → S1 | S52 → c |
| (S53, e) → S54 | S53 → eS54 |
| (S54, t) → S55 | S54 → tS55 |
| (S55, u) → S56 | S55 → uS56 |
| (S56, r) → S57 | S56 → rS57 |
| (S57, n) → S1 | S57 → n |
| (S58, r) → S59 | S58 → rS59 |
| (S59, u) → S60 | S59 → uS60 |
| (S60, e) → S1 | S60 → e |
| (S61, o) → S62 | S61 → oS62 |
| (S62, i) → S63 | S62 → iS63 |
| (S63, d) → S1 | S63 → d |
| (S64, 0) → S64  (S64, 1) → S64  (S64, 2) → S64  (S64, 3) → S64  (S64, 4) → S64  (S64, 5) → S64  (S64, 6) → S64  (S64, 7) → S64  (S64, 8) → S64  (S64, 9) → S64  (S64, .) → S66 | S64 → 0S64 | 1S64 | 2S64 | 3S64 | 4S64 | 5S64 | 6S64 | 7S64 | 8S64 | 9S64 | .S66 |
| (S65, 0) → S64  (S65, 1) → S64  (S65, 2) → S64  (S65, 3) → S64  (S65, 4) → S64  (S65, 5) → S64  (S65, 6) → S64  (S65, 7) → S64  (S65, 8) → S64  (S65, 9) → S64 | S65 → 0S64 | 1S64 | 2S64 | 3S64 | 4S64 | 5S64 | 6S64 | 7S64 | 8S64 | 9S64 |
| (S66, 0) → S64  (S66, 1) → S64  (S66, 2) → S64  (S66, 3) → S64  (S66, 4) → S64  (S66, 5) → S64  (S66, 6) → S64  (S66, 7) → S64  (S66, 8) → S64  (S66, 9) → S64 | S66 → 0S64 | 1S64 | 2S64 | 3S64 | 4S64 | 5S64 | 6S64 | 7S64 | 8S64 | 9S64 |
| (S67, 0) → S2  (S67, 1) → S2  (S67, 2) → S2  (S67, 3) → S2  (S67, 4) → S2  (S67, 5) → S2  (S67, 6) → S2  (S67, 7) → S2  (S67, 8) → S2  (S67, 9) → S2  (S67, -) → S68  (S67, +) → S68 | S67 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | -S68 | + S68 |
| (S68, 0) → S2  (S68, 1) → S2  (S68, 2) → S2  (S68, 3) → S2  (S68, 4) → S2  (S68, 5) → S2  (S68, 6) → S2  (S68, 7) → S2  (S68, 8) → S2  (S68, 9) → S2 | S68 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ РАБОТЫ

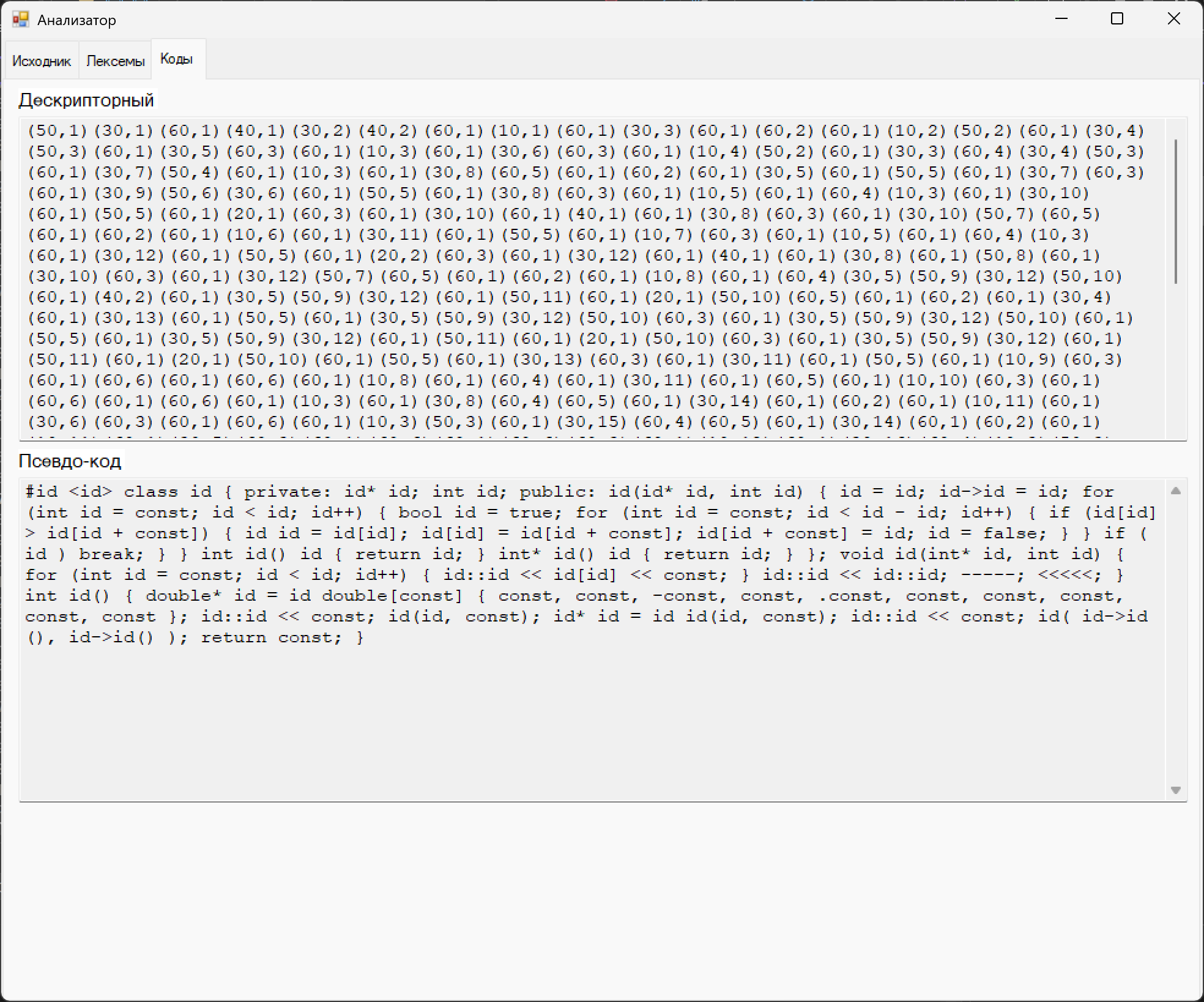


Рис. 1. Демонстрация построения кодов

В результате лабораторной работы получена программа, моделирующая работу лексического анализатора, удаляющая лишние символы из программного текста, отделяющая 6 классов лексем по таблицам и строящая дескрипторный и псевдокод по этим таблицам.

Часть исходного текста программы:

/\*

\* Метод, сбрасывающий интерфейс программы в исходное состояние

\*/

System::Void Analize::GUI::clear() {

this->outBox->Text = "";

this->textBoxPseudo->Text = "";

this->textBoxDescript->Text = "";

array<DataGridView^>^ grids = {

this->dataGridViewConst,

this->dataGridViewKeys,

this->dataGridViewIDs,

this->dataGridViewDelims,

this->dataGridViewRelative,

this->dataGridViewSigns

};

for each (DataGridView ^ table in grids) {

table->RowCount = 1;

}

}

/\*

\* Метод удаляющий пробелы, переносы и комментарии из

\* текста программы, используя анализатор

\*/

System::Void Analize::GUI::stripSource() {

analyser->clearState();

for (int i = 0; i < sourceBox->Text->Length; i++) {

char out = analyser->space\_filter(sourceBox->Text[i]);

if (out != -1) outBox->Text += gcnew System::String(&out);

}

}

/\*

\* Метод возвращающий номер лексемы в указанной таблице лексем.

\* При отстутствии лексемы в таблице вернёт 0.

\*/

int Analize::GUI::getFromTable(String^ value, DataGridView^ gridView) {

for (int i = 0; i < gridView->RowCount; i++) {

if (gridView->Rows[i]->Cells[1]->Value != nullptr)

if (gridView->Rows[i]->Cells[1]->Value->ToString() == value)

return Int32::Parse(gridView->Rows[i]->Cells[0]->Value->ToString());

}

return 0;

}

/\*

\* Метод, добавляющий лексему в таблицу лексем и на её основе

\* добавляющий лексему в дескрипторный и псевдо-код

\*/

System::Void Analize::GUI::buildCodes(int state, String^ lexem, String^ code, DataGridView^ gridView) {

if (!getFromTable(lexem, gridView)) {

int k\_count = gridView->RowCount++;

gridView->Rows[k\_count - 1]->Cells[0]->Value = k\_count.ToString();

gridView->Rows[k\_count - 1]->Cells[1]->Value = lexem;

gridView->Rows[k\_count - 1]->Cells[2]->Value = code;

}

textBoxDescript->Text += "(" + state \* 10 + "," + getFromTable(lexem, gridView) + ")";

textBoxPseudo->Text += code;

}

/\*

\* Функция разделяющая лексемы по таблицам

\*/

System::Void Analize::GUI::addToTable(int state, String^ lexem) {

lexem = lexem->Replace("\n", "\\n")->Replace("\r", "\\r");

switch (state) {

case 1: { // Ключевое слово

buildCodes(state, lexem, lexem, dataGridViewKeys);

break;

}

case 2: { // Константа

buildCodes(state, lexem, "const", dataGridViewConst);

break;

}

case 3: { // Идентификатор

buildCodes(state, lexem, "id", dataGridViewIDs);

break;

}

case 4: { // Оператор отношения

buildCodes(state, lexem, lexem, dataGridViewRelative);

break;

}

case 5: { // Знак операции

buildCodes(state, lexem, lexem, dataGridViewSigns);

break;

}

case 6: { // Разделитель

buildCodes(state, lexem, lexem, dataGridViewDelims);

break;

}

}

}

/\*

\* Обработка нажатия на кнопку "Отрыть файл"

\*/

System::Void Analize::GUI::openFileButton\_Click(System::Object^ sender, System::EventArgs^ e) {

this->openFileDialog1->FileName = "";

this->openFileDialog1->ShowDialog();

this->fileNameBox->Text = openFileDialog1->FileName;

this->sourceBox->Text = "";

StreamReader^ reader = gcnew StreamReader(openFileDialog1->FileName, Encoding::GetEncoding("windows-1251"));

do {

this->sourceBox->Text += reader->ReadLine() + " \r\n";

} while (reader->Peek() != -1);

this->sourceBox->BackColor = System::Drawing::SystemColors::Window;

}

/\*

\* Обработка нажатия на кнопку "Обработать"

\*/

System::Void Analize::GUI::processButton\_Click(System::Object^ sender, System::EventArgs^ e) {

clear(); // Сброс программы

// Удаление пробелов и переносов

this->outBox->BackColor = System::Drawing::SystemColors::Window;

stripSource();

// Лексический анализ

analyser->clearState();

char prev = this->outBox->Text[0];

for (int i = 1; i < this->outBox->Text->Length; i++) {

std::string lexem = analyser->lexem\_filter(prev, this->outBox->Text[i]);

if (!lexem.empty()) {

this->addToTable(analyser->getState(), gcnew String(lexem.c\_str()));

analyser->clearState();

}

prev = this->outBox->Text[i];

}

}

/\*

\* Метод, возвращающий является ли разделителем

\* заданный символ

\*/

bool Analyser::is\_delim(char c) {

/\*if (check\_next) {

Analyser\* checker = new Analyser();

checker->check\_next = false;

checker->lexem\_filter(c, c);

int state = checker->state;

delete checker;

return state == 6;

}

return false;\*/

std::string delimers = " ,:;(){}'\"\n\0";

return std::count(delimers.begin(), delimers.end(), c) > 0;

}

/\*

\* Метод, возвращающий является ли знаком

\* заданный символ

\*/

bool Analyser::is\_sign(char c) {

/\*if (check\_next) {

Analyser\* checker = new Analyser();

checker->check\_next = false;

checker->lexem\_filter(c, c);

int state = checker->state;

delete checker;

return (state == 4 || state == 5);

}

return false;\*/

std::string signs = ".:[]<>+-/\*=#&";

return std::count(signs.begin(), signs.end(), c) > 0;

}

/\*

\* Метод, возвращающий входит ли следующий символ в валидную лексему

\*/

bool Analyser::is\_next\_valid(char c) {

if (check\_next) {

Analyser\* checker = new Analyser(this);

checker->check\_next = false;

checker->lexem\_filter(c, c);

int state = checker->state;

delete checker;

return state > 0 && state <= 6 && state != 3;

}

return true;

}

/\*

\* Метод, возвращающий является ли знаком

\* заданный символ по состояниям

\*/

bool Analyser::is\_multisign() {

std::vector<int> n\_states = {

11, 13, 16, 21, 23, 25, 27,

29, 33, 37, 40, 43, 44, 46,

48, 49, 51, 52, 53

};

return std::count(n\_states.begin(), n\_states.end(), state) == 0;

}

Analyser::Analyser() {

this->buffer = "";

this->is\_id = false;

this->state = 0;

}

Analyser::Analyser(Analyser\* copy) {

this->buffer = copy->buffer;

this->is\_id = copy->is\_id;

this->state = copy->state;

this->check\_next = copy->check\_next;

}

/\*

\* Метод, сбрасывающий состояние конечного автомата

\*/

void Analyser::clearState() {

this->state = 0;

this->buffer = "";

this->is\_id = false;

}

/\*

\* Метод, возвращающий состояние конечного автомата

\*/

int Analyser::getState() const {

return this->state;

}

/\*

\* Метод, устанавливающий состояние конечного автомата

\*/

void Analyser::setState(int state) {

this->state = state;

}

/\*

\* Метод, лексического анализатора для удаления

\* пробелов, переносов, комментариев

\*/

char Analyser::space\_filter(char c) {

if (c == ' ') return -1;

switch (state) {

case 0: {

switch (c) {

case '/': { state = 1; break; }

case ' ': { state = 4; break; }

case '\n': { state = 5; break; }

case '\r': { state = 5; break; }

default: { state = 0; break; }

} break;

}

case 1: {

switch (c) {

case '/': { state = 1; break; }

case '\*': { state = 2; break; }

case '\n': { state = 5; break; }

case '\r': { state = 5; break; }

} break;

}

case 2: {

switch (c) {

case '\*': { state = 3; break; }

default: { state = 2; break; }

} break;

}

case 3: {

switch (c) {

case '/': { state = 5; break; }

default: { state = 2; break; }

} break;

}

case 4: {

switch (c) {

case '/': { state = 1; break; }

case ' ': { state = 5; break; }

case '\n': { state = 5; break; }

case '\r': { state = 5; break; }

default: { state = 0; break; }

} break;

}

case 5: {

switch (c) {

case '/': { state = 1; break; }

case ' ': { state = 5; break; }

case '\n': { state = 5; break; }

case '\r': { state = 5; break; }

default: { state = 0; break; }

} break;

}

}

// Печать обрабатываемых символов в консоль (отладка)

#ifdef \_DEBUG

if (c == '\n') std::cout << "\\n" << " " << state;

else if (c == '\r') std::cout << "\\r" << " " << state;

else std::cout << c << " " << state;

if (state == 0 || state == 4) {

std::cout << " - Valid";

};

std::cout << std::endl;

#endif

if (state == 0 || state == 4)

return c;

return -1;

}

/\*

\* Метод, лексического анализатора для разделения

\* лексем и определения их типа

\*/

std::string Analyser::lexem\_filter(char c, char next) {

switch (state) {

case 0: {

switch (c) {

// Ключевые слова

case 'a': { state = 61; break; }

case 'b': { state = 64; break; }

case 'c': { state = 70; break; }

case 'd': { state = 78; break; }

case 'e': { state = 86; break; }

case 'f': { state = 89; break; }

case 'i': { state = 94; break; }

case 'p': { state = 96; break; }

case 'r': { state = 106; break; }

case 't': { state = 111; break; }

case 'v': { state = 114; break; }

// Константы

case '0': { state = 2; break; }

case '1': { state = 2; break; }

case '2': { state = 2; break; }

case '3': { state = 2; break; }

case '4': { state = 2; break; }

case '5': { state = 2; break; }

case '6': { state = 2; break; }

case '7': { state = 2; break; }

case '8': { state = 2; break; }

case '9': { state = 2; break; }

// Операторы отношения

case '<': { state = 29; break; }

case '>': { state = 33; break; }

// Знаки операций

case '=': { state = 11; break; }

case '+': { state = 13; break; }

case '-': { state = 16; break; }

case '\*': { state = 21; break; }

case '/': { state = 23; break; }

case '%': { state = 25; break; }

case '!': { state = 27; break; }

case '&': { state = 37; break; }

case '|': { state = 40; break; }

case '~': { state = 43; break; }

case '^': { state = 44; break; }

case '.': { state = 46; break; }

case ',': { state = 48; break; }

case ':': { state = 49; break; }

case '[': { state = 51; break; }

case ']': { state = 52; break; }

case '#': { state = 53; break; }

// Разделители

case ' ': { state = 6; break; }

case ';': { state = 6; break; }

case '{': { state = 6; break; }

case '}': { state = 6; break; }

case '(': { state = 6; break; }

case ')': { state = 6; break; }

case '\n': { state = 6; break; }

// Строки

case '\'': { state = 7; break; }

case '\"': { state = 8; break; }

default: { state = 0; break; }

} break;

}

case 1: { // Ключевые слова

switch (c) {

case 'u': { state = 79; break; }

default: { state = 0; break; }

} break;

}

case 2: { // Константы

switch (c) {

case '0': { state = 2; break; }

case '1': { state = 2; break; }

case '2': { state = 2; break; }

case '3': { state = 2; break; }

case '4': { state = 2; break; }

case '5': { state = 2; break; }

case '6': { state = 2; break; }

case '7': { state = 2; break; }

case '8': { state = 2; break; }

case '9': { state = 2; break; }

case 'e': { state = 4; break; }

case 'E': { state = 4; break; }

case '.': { state = 3; break; }

default: { state = 0; break; }

} break;

}

case 3: {

switch (c) {

case '0': { state = 2; break; }

case '1': { state = 2; break; }

case '2': { state = 2; break; }

case '3': { state = 2; break; }

case '4': { state = 2; break; }

case '5': { state = 2; break; }

case '6': { state = 2; break; }

case '7': { state = 2; break; }

case '8': { state = 2; break; }

case '9': { state = 2; break; }

default: { state = 0; break; }

} break;

}

case 4: {

switch (c) {

case '0': { state = 9; break; }

case '1': { state = 9; break; }

case '2': { state = 9; break; }

case '3': { state = 9; break; }

case '4': { state = 9; break; }

case '5': { state = 9; break; }

case '6': { state = 9; break; }

case '7': { state = 9; break; }

case '8': { state = 9; break; }

case '9': { state = 9; break; }

case '-': { state = 5; break; }

case '+': { state = 5; break; }

default: { state = 0; break; }

} break;

}

case 5: {

switch (c) {

case '0': { state = 9; break; }

case '1': { state = 9; break; }

case '2': { state = 9; break; }

case '3': { state = 9; break; }

case '4': { state = 9; break; }

case '5': { state = 9; break; }

case '6': { state = 9; break; }

case '7': { state = 9; break; }

case '8': { state = 9; break; }

case '9': { state = 9; break; }

default: { state = 0; break; }

} break;

}

case 6: { // Разделители

switch (c) {

default: { state = 0; break; }

} break;

}

case 7: { // Строковый литерал ''

switch (c) {

case '\'': { state = 2; break; }

default: { state = 7; break; }

} break;

}

case 8: { // Строковый литерал ""

switch (c) {

case '\"': { state = 2; break; }

default: { state = 8; break; }

} break;

}

case 9: { //Константа вида xE-xx

switch (c) {

case '0': { state = 9; break; }

case '1': { state = 9; break; }

case '2': { state = 9; break; }

case '3': { state = 9; break; }

case '4': { state = 9; break; }

case '5': { state = 9; break; }

case '6': { state = 9; break; }

case '7': { state = 9; break; }

case '8': { state = 9; break; }

case '9': { state = 9; break; }

default: { state = 0; break; }

} break;

}

case 10: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 11: {

switch (c) {

case '=': { state = 12; break; }

default: { state = 0; break; }

} break;

}

case 12: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 13: {

switch (c) {

case '+': { state = 14; break; }

case '=': { state = 15; break; }

default: { state = 0; break; }

} break;

}

case 14: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 15: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 16: {

switch (c) {

case '-': { state = 17; break; }

case '=': { state = 18; break; }

case '>': { state = 19; break; }

default: { state = 0; break; }

} break;

}

case 17: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 18: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 19: {

switch (c) {

case '\*': { state = 20; break; }

default: { state = 0; break; }

} break;

}

case 20: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 21: {

switch (c) {

case '=': { state = 22; break; }

default: { state = 0; break; }

} break;

}

case 22: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 23: {

switch (c) {

case '=': { state = 24; break; }

default: { state = 0; break; }

} break;

}

case 24: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 25: {

switch (c) {

case '=': { state = 26; break; }

default: { state = 0; break; }

} break;

}

case 26: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 27: {

switch (c) {

case '=': { state = 28; break; }

default: { state = 0; break; }

} break;

}

case 28: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 29: {

switch (c) {

case '=': { state = 30; break; }

case '<': { state = 31; break; }

default: { state = 0; break; }

} break;

}

case 30: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 31: {

switch (c) {

case '=': { state = 32; break; }

default: { state = 0; break; }

} break;

}

case 32: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 33: {

switch (c) {

case '=': { state = 34; break; }

case '>': { state = 35; break; }

default: { state = 0; break; }

} break;

}

case 34: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 35: {

switch (c) {

case '=': { state = 36; break; }

default: { state = 0; break; }

} break;

}

case 36: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 37: {

switch (c) {

case '&': { state = 38; break; }

case '=': { state = 38; break; }

default: { state = 0; break; }

} break;

}

case 38: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 39: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 40: {

switch (c) {

case '|': { state = 41; break; }

case '=': { state = 42; break; }

default: { state = 0; break; }

} break;

}

case 41: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 42: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 43: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 44: {

switch (c) {

case '=': { state = 45; break; }

default: { state = 0; break; }

} break;

}

case 45: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 46: {

switch (c) {

case '\*': { state = 47; break; }

default: { state = 0; break; }

} break;

}

case 47: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 48: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 49: {

switch (c) {

case ':': { state = 50; break; }

default: { state = 0; break; }

} break;

}

case 50: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 51: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 52: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 53: {

switch (c) {

default: { state = 0; break; }

} break;

}

case 54: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 55: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 56: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 57: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 58: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 59: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 60: { // Не занято

switch (c) {

default: { state = 0; break; }

} break;

}

case 61: {

switch (c) {

case 'u': { state = 62; break; }

default: { state = 0; break; }

} break;

}

case 62: {

switch (c) {

case 't': { state = 63; break; }

default: { state = 0; break; }

} break;

}

case 63: {

switch (c) {

case 'o': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 64: {

switch (c) {

case 'r': { state = 65; break; }

case 'o': { state = 68; break; }

default: { state = 0; break; }

} break;

}

case 65: {

switch (c) {

case 'e': { state = 66; break; }

default: { state = 0; break; }

} break;

}

case 66: {

switch (c) {

case 'a': { state = 67; break; }

default: { state = 0; break; }

} break;

}

case 67: {

switch (c) {

case 'k': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 68: {

switch (c) {

case 'o': { state = 69; break; }

default: { state = 0; break; }

} break;

}

case 69: {

switch (c) {

case 'l': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 70: {

switch (c) {

case 'l': { state = 71; break; }

case 'h': { state = 74; break; }

case 'a': { state = 76; break; }

default: { state = 0; break; }

} break;

}

case 71: {

switch (c) {

case 'a': { state = 72; break; }

default: { state = 0; break; }

} break;

}

case 72: {

switch (c) {

case 's': { state = 73; break; }

default: { state = 0; break; }

} break;

}

case 73: {

switch (c) {

case 's': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 74: {

switch (c) {

case 'a': { state = 75; break; }

default: { state = 0; break; }

} break;

}

case 75: {

switch (c) {

case 'r': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 76: {

switch (c) {

case 's': { state = 77; break; }

default: { state = 0; break; }

} break;

}

case 77: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 78: {

switch (c) {

case 'o': { state = 1; break; }

case 'e': { state = 82; break; }

default: { state = 0; break; }

} break;

}

case 79: {

switch (c) {

case 'b': { state = 80; break; }

default: { state = 0; break; }

} break;

}

case 80: {

switch (c) {

case 'l': { state = 81; break; }

default: { state = 0; break; }

} break;

}

case 81: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 82: {

switch (c) {

case 'l': { state = 83; break; }

default: { state = 0; break; }

} break;

}

case 83: {

switch (c) {

case 'e': { state = 84; break; }

default: { state = 0; break; }

} break;

}

case 84: {

switch (c) {

case 't': { state = 85; break; }

default: { state = 0; break; }

} break;

}

case 85: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 86: {

switch (c) {

case 'l': { state = 87; break; }

default: { state = 0; break; }

} break;

}

case 87: {

switch (c) {

case 's': { state = 88; break; }

default: { state = 0; break; }

} break;

}

case 88: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 89: {

switch (c) {

case 'a': { state = 90; break; }

case 'o': { state = 93; break; }

default: { state = 0; break; }

} break;

}

case 90: {

switch (c) {

case 'l': { state = 91; break; }

default: { state = 0; break; }

} break;

}

case 91: {

switch (c) {

case 's': { state = 92; break; }

default: { state = 0; break; }

} break;

}

case 92: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 93: {

switch (c) {

case 'r': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 94: {

switch (c) {

case 'n': { state = 95; break; }

case 'f': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 95: {

switch (c) {

case 't': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 96: {

switch (c) {

case 'r': { state = 97; break; }

case 'u': { state = 102; break; }

default: { state = 0; break; }

} break;

}

case 97: {

switch (c) {

case 'i': { state = 98; break; }

default: { state = 0; break; }

} break;

}

case 98: {

switch (c) {

case 'v': { state = 99; break; }

default: { state = 0; break; }

} break;

}

case 99: {

switch (c) {

case 'a': { state = 100; break; }

default: { state = 0; break; }

} break;

}

case 100: {

switch (c) {

case 't': { state = 101; break; }

default: { state = 0; break; }

} break;

}

case 101: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 102: {

switch (c) {

case 'b': { state = 103; break; }

default: { state = 0; break; }

} break;

}

case 103: {

switch (c) {

case 'l': { state = 104; break; }

default: { state = 0; break; }

} break;

}

case 104: {

switch (c) {

case 'i': { state = 105; break; }

default: { state = 0; break; }

} break;

}

case 105: {

switch (c) {

case 'c': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 106: {

switch (c) {

case 'e': { state = 107; break; }

default: { state = 0; break; }

} break;

}

case 107: {

switch (c) {

case 't': { state = 108; break; }

default: { state = 0; break; }

} break;

}

case 108: {

switch (c) {

case 'u': { state = 109; break; }

default: { state = 0; break; }

} break;

}

case 109: {

switch (c) {

case 'r': { state = 110; break; }

default: { state = 0; break; }

} break;

}

case 110: {

switch (c) {

case 'n': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 111: {

switch (c) {

case 'r': { state = 112; break; }

default: { state = 0; break; }

} break;

}

case 112: {

switch (c) {

case 'u': { state = 113; break; }

default: { state = 0; break; }

} break;

}

case 113: {

switch (c) {

case 'e': { state = 1; break; }

default: { state = 0; break; }

} break;

}

case 114: {

switch (c) {

case 'o': { state = 115; break; }

default: { state = 0; break; }

} break;

}

case 115: {

switch (c) {

case 'i': { state = 116; break; }

default: { state = 0; break; }

} break;

}

case 116: {

switch (c) {

case 'd': { state = 1; break; }

default: { state = 0; break; }

} break;

}

}

buffer += c;

if (state == 0)

is\_id = true; // Если автомат перешёл в состояние S=0, то вся лексема - иденификатор

// Ключевые слова и константы

if ((state == 1 || state == 2 || state == 9) && (is\_delim(next) || is\_sign(next)) && !is\_id) {

if ((state != 2 && state != 9) || next != '.') {

if (state == 9) state = 2;

return buffer;

}

}

// Знаки

if ((state > 10 && state <= 60) && (!is\_next\_valid(next) || is\_multisign()) && !is\_id) {

if ((state >= 28 && state <= 30) || (state >= 33 && state <= 34) || state == 12) {

state = 4;

return buffer;

}

state = 5;

return buffer;

}

// Разделители

if (state == 6) {

return buffer;

}

// Идентификаторы

if (buffer != "" && (is\_id || state > 60 || (state >= 3 && state <= 5)) && (is\_delim(next) || (is\_sign(next) && !is\_next\_valid(next)))) {

state = 3; return buffer;

}

return "";

}

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы было выполнено построение дескрипторного кода и псевдокода по построенным ранее таблицам лексем.

Функция лексического анализатора оформлена в виде графического приложения, и её работа протестирована на некотором тексте программы. Входной текст программы представлял собой описание класса сортированного массива, а также работу с ним.

Конечный автомат, на котором была построена функция лексического анализатора всё так же состоял из 116-ти состояний, среди которых всё так же 6 конечных для определения класса лексем.

Также была построена регулярная грамматика по автомату из лабораторной работы 3-4. Она была смешная и занимала несколько страниц А4 просто потому, что может.