Институт информационных технологий

Кафедра: Математическое и программное обеспечение ЭВМ

Дисциплина: Теория автоматов и формальных языков

ЛАБОРАТОРНАЯ РАБОТА № 5-6

Тема: **Построение таблиц идентификаторов**

Выполнил:

студент гр. 1ПИб-02-2оп-23

Кринкин Олег Алексеевич

Проверил:

доцент, к.т.н. Ганичева Оксана Георгиевна

ЗАДАНИЕ

Написать часть/функцию лексического анализатора, обрабатывающую входную строку по конечному автомату на основе оператора выбора (эта часть из предыдущей работы) и **дополнить ее формированием таблиц с классами лексем.**

**Результат работы программы (интерфейс):**

**1. считываемая входная строка (код в соответствие со своим вариантом)**

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

**3. В работе предусмотреть проверку на повторное вхождение лексемы в заданный класс, отнесение к правильному классу лексем, выдача ошибки в случае неверной лексемы и др.)**

**В отчете должна быть отражена след информация.**

**1. код программы с комментариями**

**2. скриншоты с результатами работы (сделать их читаемыми по размерам). Обратить внимание на демонстрацию результатов по п.3**

ХОД РАБОТЫ

Эта лабораторная работа является расширением предыдущей, поэтому во входных данных и в автомате значительных изменений произведено не было.

1. В интерфейс программы добавлено ещё 4 таблицы для каждого из типов лексем (рис. 1).

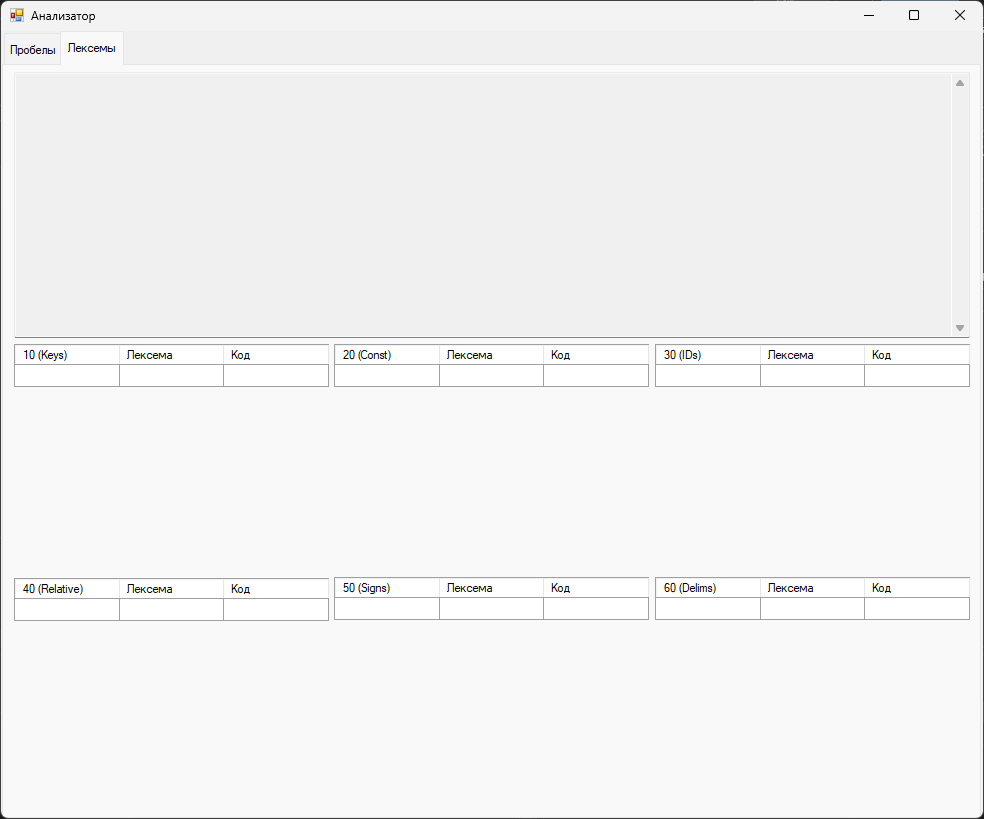


Рис. . Интерфейс со всеми таблицами

1. Лексема попадает в таблицу в зависимости от состояния автомата, во время определения отдельной лексемы. Код добавления лексемы в таблицу выглядит следующим образом:

|  |
| --- |
| System::Void addToTable(int state, String^ lexem) {  lexem = lexem->Replace("\n", "\\n")->Replace("\r", "\\r");  switch (state) {  case 1: { // Ключевое слово  if (!findInTable(lexem, dataGridViewKeys)) {  int k\_count = this->dataGridViewKeys->RowCount++; dataGridViewKeys->Rows[k\_count - 1]->Cells[0]->Value = k\_count.ToString();  dataGridViewKeys->Rows[k\_count - 1]->Cells[1]->Value = lexem;  dataGridViewKeys->Rows[k\_count - 1]->Cells[2]->Value = lexem;  }  break;  }  case 2: { // Константа  if (!findInTable(lexem, dataGridViewConst)) {  int c\_count = this->dataGridViewConst->RowCount++;  dataGridViewConst->Rows[c\_count - 1]->Cells[0]->Value = c\_count.ToString();  dataGridViewConst->Rows[c\_count - 1]->Cells[1]->Value = lexem;  dataGridViewConst->Rows[c\_count - 1]->Cells[2]->Value = "const";  }  break;  }  case 3: { // Идентификатор  if (!findInTable(lexem, dataGridViewIDs)) {  int id\_count = this->dataGridViewIDs->RowCount++;  dataGridViewIDs->Rows[id\_count - 1]->Cells[0]->Value = id\_count.ToString();  dataGridViewIDs->Rows[id\_count - 1]->Cells[1]->Value = lexem;  dataGridViewIDs->Rows[id\_count - 1]->Cells[2]->Value = "id";  }  break;  }  case 4: { // Оператор отношения  if (!findInTable(lexem, dataGridViewRelative)) {  int r\_count = this->dataGridViewRelative->RowCount++;  dataGridViewRelative->Rows[r\_count - 1]->Cells[0]->Value = r\_count.ToString();  dataGridViewRelative->Rows[r\_count - 1]->Cells[1]->Value = lexem;  dataGridViewRelative->Rows[r\_count - 1]->Cells[2]->Value = lexem;  }  break;  }  case 5: { // Знак операции  if (!findInTable(lexem, dataGridViewSigns)) {  int s\_count = this->dataGridViewSigns->RowCount++;  dataGridViewSigns->Rows[s\_count - 1]->Cells[0]->Value = s\_count.ToString();  dataGridViewSigns->Rows[s\_count - 1]->Cells[1]->Value = lexem;  dataGridViewSigns->Rows[s\_count - 1]->Cells[2]->Value = lexem;  }  break;  }  case 6: { // Разделитель  if (!findInTable(lexem, dataGridViewDelims)) {  int d\_count = this->dataGridViewDelims->RowCount++;  dataGridViewDelims->Rows[d\_count - 1]->Cells[0]->Value = d\_count.ToString();  dataGridViewDelims->Rows[d\_count - 1]->Cells[1]->Value = lexem;  dataGridViewDelims->Rows[d\_count - 1]->Cells[2]->Value = lexem;  }  break;  }  }  } |

1. Перед добавлением элемента проверяется его присутствие в таблице следующей функцией:

|  |
| --- |
| // Функция возвращающая есть ли в таблице указанный элемент  bool findInTable(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 true;  }  return false;  } |

1. Блок схема добавления лексем в таблицы представлена на рис. 2. Идентификаторы определяются методом исключения: если найден разделитель, но состояние автомата не попадает в диапазон 1-6, то автомат переходит в состояние 3, и лексема возвращается (как видно из блок-схемы таким образом идентификатор окажется записан в таблицу идентификаторов, в соответствие состоянию 3). На данном этапе ошибки в написании идентификаторов не обнаруживаются.
2. Определение операторов происходит довольно просто – из состояния 0 по символу оператора автомат переходит в состояние 4 или 5 (в зависимости от класса оператора), которые являются конечными, однако с операторами, состоящими из двух символов алгоритм сложнее – перед тем как выйти из конечного состояния, автомат проверяет, какой именно разделитель идёт далее. Если разделителем оказался не пробел, а знак, значит текущая лексема составной знак и возврат из конечного состояния не происходит. В виде графа это представлено на рис. 3.

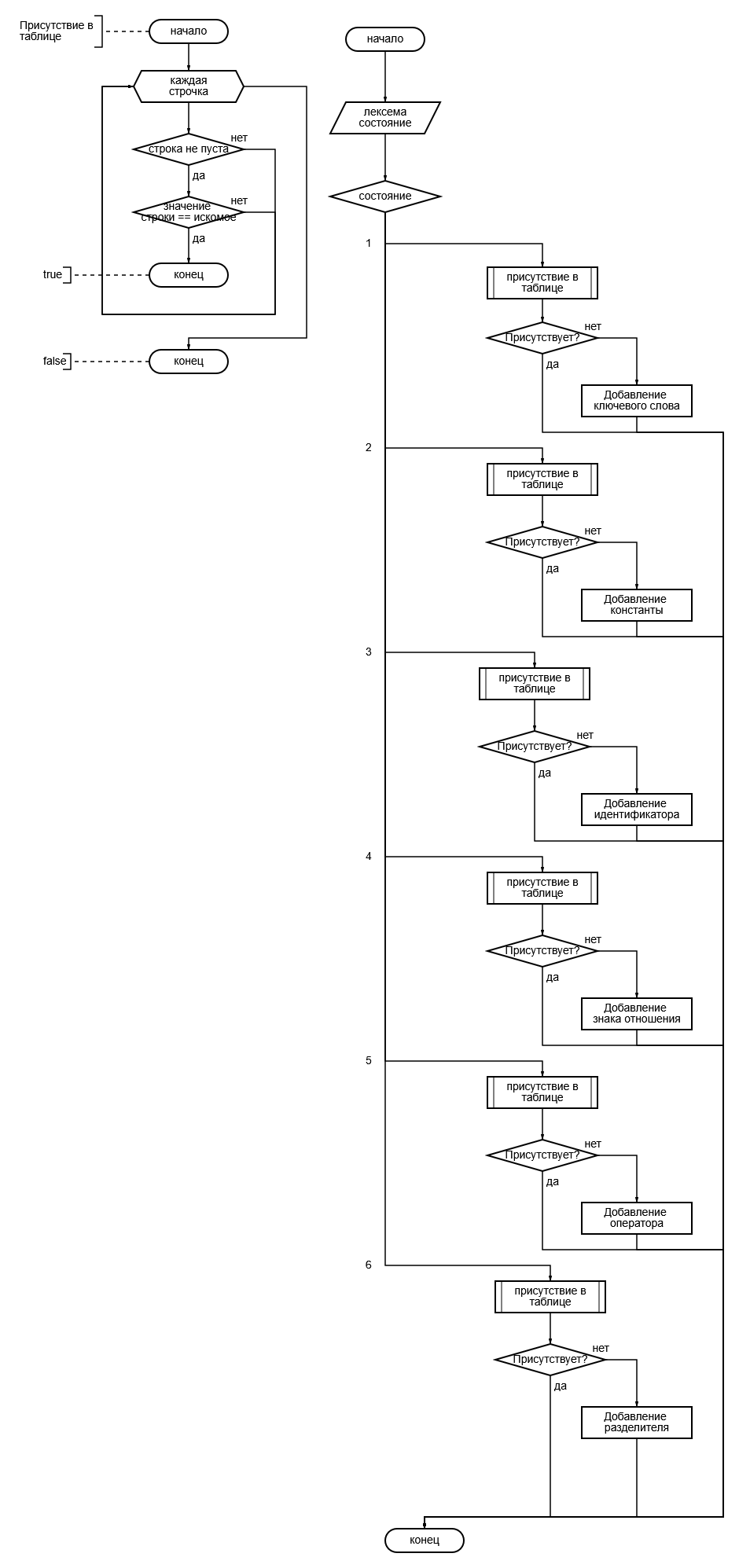


Рис. . Блок-схема алгоритма добавления в таблицу

Изображение выглядит как диаграмма, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

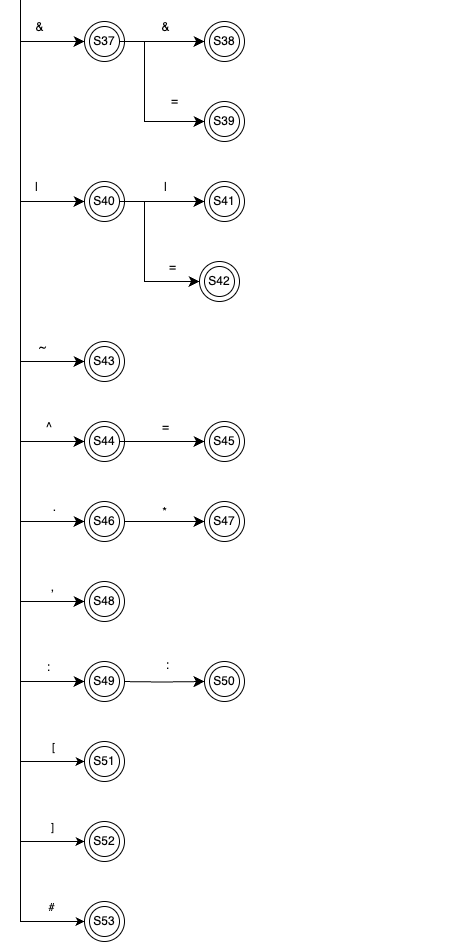


Рис. . Обработка составных знаков

РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ РАБОТЫ

В результате выполнения лабораторной работы в ранее созданной программе (рис. 4), моделирующей работу лексического анализатора, было сделано распознавание и занесение в таблицу всех классов лексем, включая операторы, разделители и идентификаторы. Пример работы программы также представлен на рис. 4.

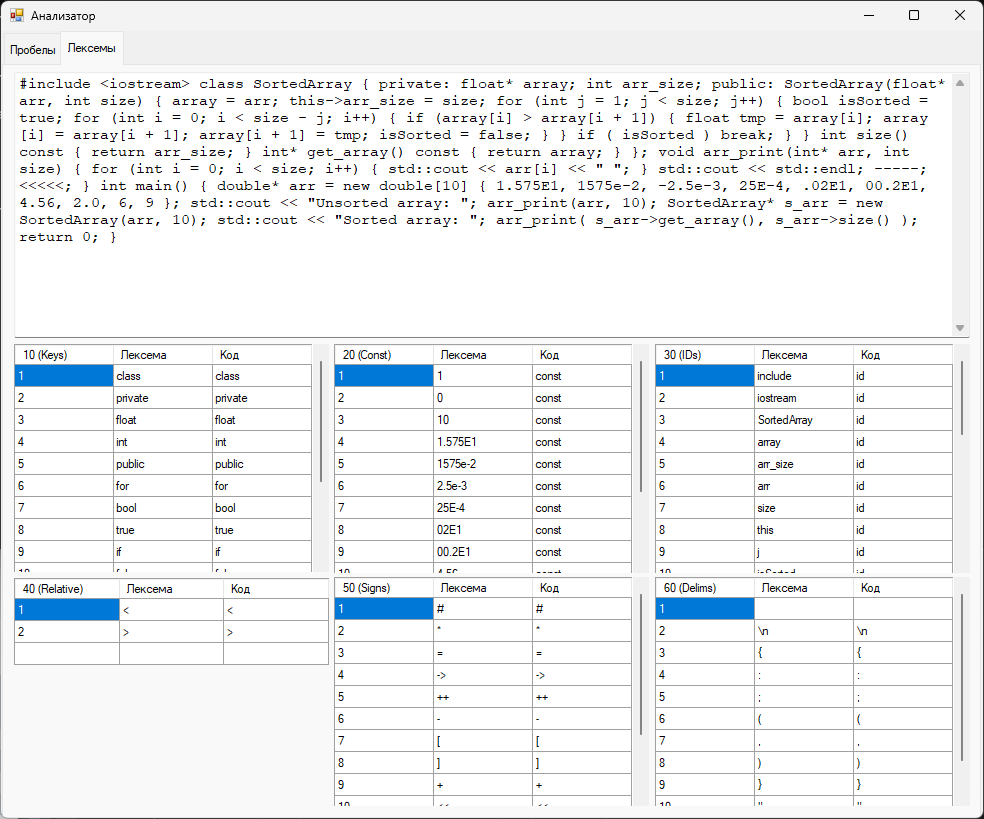


Рис. 4. Окно программы

Для полного тестирования был использован следующий набор лексем:

|  |
| --- |
| 314.56  31.56E+02  32.33E  145.16A7  <<<<<  ++-+==  3\_nomer  nomer\_3  4nomer  --=+<=>=>>  ;;;  int\_1  int char  int\_char  aq, |

После его обработки лексическим анализатором получен следующий результат (рис. 5):

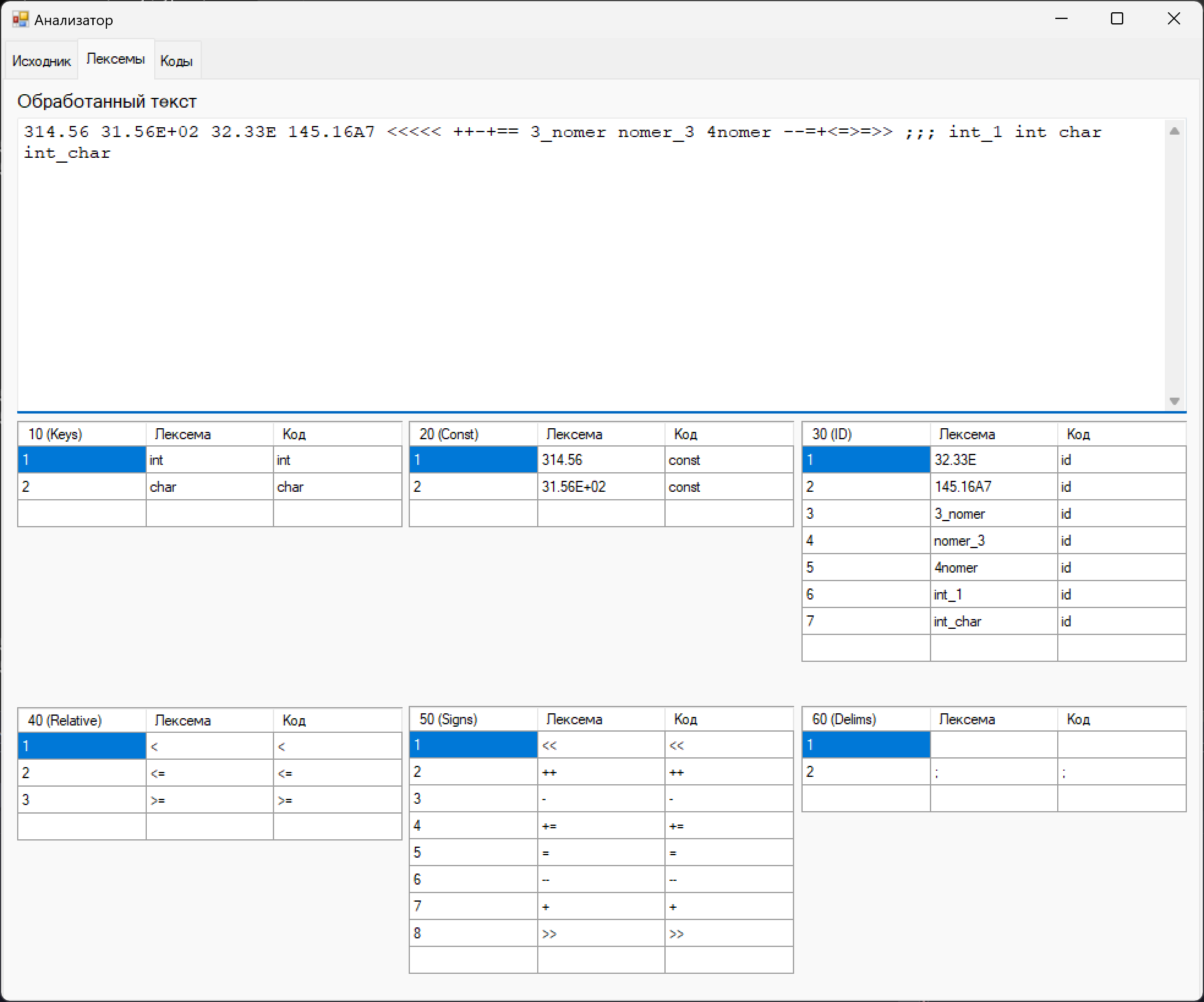


Рис. . Результат анализа тестового фрагмента

Часть программного кода спроектированной программы:

private:

// Cчётчик строк

int line = 0;

// Функция добавляющая в ноер строки в исходный текст

String^ addLineNum(String^ target, int count) {

line = 0;

count = (count + 1).ToString()->Length;

target = ++line + spaces(count - 1) + " " + target;

String^ out = gcnew String("");

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

if (target[i] == '\n') {

out += "\n" + ++line + spaces(count - (line.ToString()->Length)) + " ";

continue;

}

out += target[i];

}

target = out;

return target;

}

// Функция преобразующая System::String в std::string

std::string to\_string(String^ string) {

msclr::interop::marshal\_context context;

return context.marshal\_as<std::string>(string);

}

// Функция возвращающая некоторое количество пробелов

String^ spaces(int count) {

String^ spaces = gcnew String("");

for (int i = 0; i < count; i++)

spaces += " ";

return spaces;

}

// Функция возвращающая есть ли в таблице указанный элемент

bool findInTable(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 true;

}

return false;

}

// Функция разделяющая лексемы по таблицам

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

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

switch (state) {

case 1: { // Ключевое слово

if (!findInTable(lexem, dataGridViewKeys)) {

int k\_count = this->dataGridViewKeys->RowCount++;

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

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

dataGridViewKeys->Rows[k\_count - 1]->Cells[2]->Value = lexem;

}

break;

}

case 2: { // Константа

if (!findInTable(lexem, dataGridViewConst)) {

int c\_count = this->dataGridViewConst->RowCount++;

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

dataGridViewConst->Rows[c\_count - 1]->Cells[1]->Value = lexem;

dataGridViewConst->Rows[c\_count - 1]->Cells[2]->Value = "const";

}

break;

}

case 3: { // Идентификатор

if (!findInTable(lexem, dataGridViewIDs)) {

int id\_count = this->dataGridViewIDs->RowCount++;

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

dataGridViewIDs->Rows[id\_count - 1]->Cells[1]->Value = lexem;

dataGridViewIDs->Rows[id\_count - 1]->Cells[2]->Value = "id";

}

break;

}

case 4: { // Оператор отношения

if (!findInTable(lexem, dataGridViewRelative)) {

int r\_count = this->dataGridViewRelative->RowCount++;

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

dataGridViewRelative->Rows[r\_count - 1]->Cells[1]->Value = lexem;

dataGridViewRelative->Rows[r\_count - 1]->Cells[2]->Value = lexem;

}

break;

}

case 5: { // Знак операции

if (!findInTable(lexem, dataGridViewSigns)) {

int s\_count = this->dataGridViewSigns->RowCount++;

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

dataGridViewSigns->Rows[s\_count - 1]->Cells[1]->Value = lexem;

dataGridViewSigns->Rows[s\_count - 1]->Cells[2]->Value = lexem;

}

break;

}

case 6: { // Разделитель

if (!findInTable(lexem, dataGridViewDelims)) {

int d\_count = this->dataGridViewDelims->RowCount++;

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

dataGridViewDelims->Rows[d\_count - 1]->Cells[1]->Value = lexem;

dataGridViewDelims->Rows[d\_count - 1]->Cells[2]->Value = lexem;

}

break;

}

}

}

// Обработка нажатия на кнопку "Отрыть файл"

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

this->openFileDialog1->FileName = "";

this->openFileDialog1->ShowDialog();

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

std::string& source = analyser->readSource(to\_string(openFileDialog1->FileName));

this->sourceBox->Text = gcnew String(source.c\_str());

this->sourceBox->Text = addLineNum(this->sourceBox->Text, analyser->getLineCount(source));

this->sourceBox->Text = this->sourceBox->Text->Replace("\n", "\r\n");

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

}

// Обработка нажатия на кнопку "Обработать"

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

// Удаление пробелов и переносов

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

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

const char\* stripped = analyser->stripSource().c\_str();

this->outBox->Text = gcnew String(stripped);

this->outBoxL->Text = this->outBox->Text->Replace("\n", " ");;

this->outBox->Text = addLineNum(this->outBox->Text, analyser->getLineCount(stripped));

this->outBox->Text = this->outBox->Text->Replace("\n", "\r\n");

this->linesNum->Text = analyser->getLineCount(stripped).ToString();

// Лексический анализ

const std::string text = analyser->getStripped();

analyser->clearState();

char prev = text[0];

for (int i = 1; i < text.size(); i++) {

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

if (!lexem.empty()) {

#ifdef \_DEBUG

std::cout << "'" << lexem << "' - " << analyser->getState() << std::endl;

#endif

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

analyser->clearState();

}

prev = text[i];

}

}

System::Void GUI\_Load(System::Object^ sender, System::EventArgs^ e) {

// Инициализация таблиц

array<DataGridView^>^ tables = {

this->dataGridViewKeys,

this->dataGridViewConst,

this->dataGridViewIDs,

this->dataGridViewRelative,

this->dataGridViewSigns,

this->dataGridViewDelims

};

for each (DataGridView^ table in tables) {

table->ColumnCount = 3;

table->Columns[1]->HeaderCell->Value = "Лексема";

table->Columns[2]->HeaderCell->Value = "Код";

}

this->dataGridViewKeys->Columns[0]->HeaderCell->Value = "10 (Keys)";

this->dataGridViewConst->Columns[0]->HeaderCell->Value = "20 (Const)";

this->dataGridViewIDs->Columns[0]->HeaderCell->Value = "30 (IDs)";

this->dataGridViewRelative->Columns[0]->HeaderCell->Value = "40 (Relative)";

this->dataGridViewSigns->Columns[0]->HeaderCell->Value = "50 (Signs)";

this->dataGridViewDelims->Columns[0]->HeaderCell->Value = "60 (Delims)";

}

class Analyser {

private:

std::string source;

std::string stripped;

std::string buffer = "";

bool is\_id = false;

int state = 0;

int check\_next = true;

bool 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;

}

bool 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;

}

bool 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;

}

bool 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;

}

public:

Analyser() {

this->buffer = "";

this->is\_id = false;

this->state = 0;

}

Analyser(Analyser\* copy) {

this->buffer = copy->buffer;

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

this->state = copy->state;

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

}

void Analyser::clearState() {

this->state = 0;

this->buffer = "";

this->is\_id = false;

}

int Analyser::getState() const {

return this->state;

}

void Analyser::setState(int state) {

this->state = state;

}

const std::string& getStripped() const {

return this->stripped;

}

// Чтение файла

std::string& readSource(std::string fileName) {

std::ifstream file(fileName);

std::string line;

this->source = "";

while (std::getline(file, line)) {

this->source += line + '\n';

}

file.close();

return this->source;

}

// Функция, обрабатывающая текст и оправляющая символы анализатору

std::string& stripSource() {

this->state = 0;

this->stripped = "";

for (int i = 0; i < source.size(); i++) {

char out = space\_filter(source[i]);

if (out != -1) this->stripped += out;

}

return this->stripped;

}

// Функция, возвращающая количество строк в тексте

int getLineCount(std::string text) {

int res = 0;

for (int i = 0; i < text.length() ;i++)

if (text[i] == '\n')

res++;

return res;

}

// Функция анализатора

char 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 = 4; break; }

default: { state = 0; break; }

} break;

}

case 1: {

switch (c) {

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

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

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

case ' ': { 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; }

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;

}

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

/\*if (state == 0) buffer = "";\*/

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

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

default: { state = 0; break; }

} break;

}

case 5: {

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 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: { // Не занято

switch (c) {

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) && (is\_delim(next) || is\_sign(next)) && !is\_id) {

if (state != 2 || next != '.')

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 разделение лексем на 6 классов, занося их в отдельные для каждого класса таблицы лексем, при этом учитывая повторное вхождение лексем в тексте, а также осуществляя их подсчёт.

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 состоял уже из 116-х состояний (в результате добавления обработки знаков), среди которых всё так же 6 конечных для определения класса лексем.