*Министерство науки и высшего образования Российской Федерации*

*Федеральное государственное бюджетное образовательное учреждение высшего образования*

*«Алтайский государственный технический университет им. И.И. Ползунова»*

*Факультет Информационных технологий*

*Кафедра Информатики, вычислительной техники и информационной безопасности*

|  |  |  |
| --- | --- | --- |
|  | *Курсовая работа защищена с оценкой\_\_\_\_\_\_\_* | |
|  | *Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Л.И. Сучкова*  *подпись* | |
|  | *«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_2021 г.* | |
|  | |  | | |
| *Курсовая работа* | | | | |
| *Разработка кроссплатформенного синтаксического анализатора модельного языка с разветвляющимися конструкциями* | | | | |
| *по дисциплине Основы лингвистического анализа*  *наименование дисциплины* | | | | |
| *КР.09.03.01.16.000 ПЗ*  *Обозначение документа* | | | | |
| *Студент группы ИВТ-72 А.А. Романюк* | | | | |
| *Преподаватель профессор, д.т.н. Л.И. Сучкова* | | | | |

*БАРНАУЛ 2021*

*Министерство науки и высшего образования Российской Федерации*

*Федеральное государственное бюджетное образовательное учреждение высшего образования*

*«Алтайский государственный технический университет им. И.И. Ползунова»*

*Факультет Информационных технологий*

*Кафедра Информатики, вычислительной техники и информационной безопасности*

ЗАДАНИЕ

на курсовую работу

По направлению 09.03.01 «Информатика и вычислительная техника»

Студенту группы ИВТ-72 Романюк Анастасии Александровне

На тему: Разработка кроссплатформенного синтаксического анализатора модельного языка с разветвляющимися конструкциями

По дисциплине «Основы лингвистического анализа»

Календарный план выполнения задания

|  |  |
| --- | --- |
| Этап разработки | Срок выполнения |
| 1. Проектирование процесса разработки языка | 1-13 марта |
| 2. Программная реализация | 14-31 марта |
| 3. Тестирование приложения | 1-8 апреля |
| 4. Оформление и сдача работы на проверку | 8-10 апреля |
| 5. Срок представления работы к защите | 12 апреля |

*Преподаватель профессор, д.т.н. Л.И. Сучкова*

*подпись*

Задание принял к исполнению:

*Студент группы ИВТ-72 А.А. Романюк*

*подпись*

 Дата выдачи задания « 1 » марта 2021 г.

Реферат

Пояснительная записка к курсовой работе по теме «Разработка кроссплатформенного синтаксического анализатора модельного языка с разветвляющимися конструкциями», состоящая из 45 листов, содержит 35 рисунков и 1 таблицу.

Курсовая работа посвящена разработке кроссплатформенного синтаксического анализатора модельного языка с разветвляющимися конструкциями.

# Содержание

[Содержание 4](#_Toc69229277)

[Введение 5](#_Toc69229278)

[1 Проектирование программы для кроссплатформенного анализатора для модельного языка с разветвляющимися конструкциями 6](#_Toc69229279)

[1.1 Построение КС грамматики языка программирования 6](#_Toc69229280)

[1.2 Программирование лексического анализатора (сканера) 8](#_Toc69229281)

[1.3 Построение синтаксических диаграмм и их разметка для программирования 9](#_Toc69229282)

[2 Программная реализация 16](#_Toc69229283)

[3 Тестирование программы 19](#_Toc69229284)

[Заключение 24](#_Toc69229285)

[Cписок использованных источников 25](#_Toc69229286)

[Приложение А. Исходный код программы 26](#_Toc69229287)

# Введение

На сегодняшний день появилась необходимость в кроссплатформенном синтасическом анализаторе. Для лучшего понимания процесса работы с модельным языком с разветвляющимися конструкциями было создано программное обеспечение, которое описывается в данной курсовой работе.

Целью данной курсовой работы является разработка программы для кроссплатформенного синтаксического анализатора модельного языка с разветвляющимися конструкциями.

# Проектирование программы для кроссплатформенного анализатора для модельного языка с разветвляющимися конструкциями

## Построение КС грамматики языка программирования

Для выполнения поставленной задачи была разработана следующая КС грамматика языка, выглядящая следующим образом:

**<программа S (аксиома)>** → <описание переменных одного типа B> ***<программа S>*** | <описание функции A> ***<программа S>*** | void main()<блок O>

**<описание функции A>** → <тип функции TF> <ID>(<параметры функции P>)<блок O>

**<тип функций** **TF>** → <тип данных T> | void

**<описание переменных одного типа B>** → <тип данных T> <список переменных L>;

**<список переменных L****>** → <ID>, ***<список переменных L>*** | <ID> <инициал I>, ***<список переменных L>*** | <ID> | <ID> <инициал I>

**<тип данных T>** → int | char

**<блок O>** → { <операторы Z> }

**<операторы Z>** → <один оператор Q> ***<операторы Z>*** | <описание переменных одного типа B> ***<операторы Z>*** | <один оператор Q> | <описание переменных одного типа B>

**<один оператор Q>** → <присваивание F>; | <вызов функции R>; | <блок условия IF>; | return <A1>;

**<присваивание F>** → <ID> <оператор присваивания E> <A1>

**<блок условия IF>** → if ( <A1> ) <блок O>

**<вызов функции R>** → <ID> ( <список аргументов функции M> )

**<параметры функции P>** → <тип данных T> <ID>, ***<параметры функции P>*** *|* <тип данных T> <ID> | eps

**<список аргументов функции M>** → <операторы A1>, ***<список аргументов функции M>*** | <операторы A1> | eps

**<инициал I>** → = <A1>

**<символьная константа CC>** → '<символ клавиатуры>'

**<целочисл константа IC>** -> <цифра>***<целочисл константа IC>***|<цифра>

**<ID>** → <символ нецифра> <символы именования Sb>

**<символы именования Sb>** → ***<символы именования Sb>***<символ нецифра> | ***<символы именования Sb>***<цифра> | eps

**<оператор присваивания E>** → += | -= | = | \*= | /=

**<символ нецифра>** → a|…|z|A|…|Z

**<цифра>** → 0|…|9

**<символ клавиатуры>** → `|…|/

**<знак сравнения SC>** → > | < | == | != | >= | <=

**<знак операции SO>** → + | - | \* | /

**<A1>** → <A2> | ***<A1>***&&<A2> | ***<A1>***||<A2> | !***<A1>***

**<A2>** →A3| ***<A2>*** <знак сравнения SC> <A3>

**<A3>** →A4| ***<A3>*** <знак операции SO> <A4>

**<A4>** → <целочисл константа IC> | <символьная константа CC> | <вызов функции R> | <ID> | ( <A1> )

## Программирование лексического анализатора (сканера)

Для того, чтобы запрограммировать лексический анализатор, необходимо определить в файле типы данных, соответствующие лексемам. Также необходимо предусмотреть процедуру выдачи сообщений об ошибке при разборе. В сканере необходимо предусмотреть:

* чтение файла с исходным кодом;
* вызов сканера до тех пор, пока не достигнут конец исходного файла;
* передача отсканированной лексемы и ее типа другим модулям.

Также, перед программированием сканера, необходимо определить, какого типа комментарии допускаются в тексте программы.

Поскольку программа подразумевает работу с С-подобными языками программамирования, то допускаются комментарии начинающиеся с символов // и /\*.

В языке должны быть предусмотрены ключевые слова, которые позволят осуществить выделенные задачи. Лексемы приведены в таблице 1.

Таблица 1 – Выделенные лексемы

|  |  |  |
| --- | --- | --- |
| **Лексема** | **Тип** | **Символ-ограничитель** |
| Id | T\_ID | не <буква>, не <цифра> |
| Ключевые слова    int  char  void  main  return  if  else | T\_INT  T\_CHAR  T\_VOID  T\_MAIN  T\_RETURN  T\_IF  T\_ELSE | Не <буква>, не <цифра>>  Не <буква>, не <цифра>  Не <буква>, не <цифра>  Не <буква>, не <цифра>  Не <буква>, не <цифра>  Не <буква>, не <цифра>  Не <буква>, не <цифра> |
| Знаки операций  =  +  -  \*  /  +=  \*=  -=  /=  !  &&  || | T\_EQ  T\_PLUS  T\_SUB  T\_MUL  T\_DIV  T\_PLUSEQ  T\_MULEQ  T\_SUBEQ  T\_DIVEQ  T\_NOT  T\_AND  T\_OR | Не =  Не =  Не =  Не =  Не =  Любой  Любой  Любой  Любой  Не =  Любой  Любой |
| Знаки сравнения  >  <  ==  !=  >=  <= | T\_MORE  T\_LESS  T\_EQUAL  T\_NOT\_EQUAL  T\_MORE\_AND\_EQUAL  T\_LESS\_AND\_EQUAL | Не =  Не =  Любой  Любой  Любой  Любой |
| Константа  Целочисленная  Cимвольная | T\_CONSTINT  T\_CONSTCHAR | Не цифра, не буква  Не цифра, не буква |
| Символы-разделители  ;  , | T\_SEMI  T\_COM | Любой  Любой |
| Скобки  (  )  {  } | T\_LEFTROUNDBR  T\_RIGHTROUNDBR  T\_LEFTBRACE  T\_RIGHTBRACE | Любой  Любой  Любой  Любой |
| # (конец программы)  Ошибка | T\_END  T\_ERROR |  |

## Построение синтаксических диаграмм и их разметка для программирования

Для проверки правильности конструкций исходной программы в соответсвии с правилами грамматики, описанными в пункте 1.1, необходимо разработать синтаксический анализатор.

В ходе проектирования языка были выделены следующие диаграммы:

* <Аксиома S> (рисунок 1);
* <Описание функции A> (рисунок 2);
* <Тип данных T> (рисунок 3);
* <Тип функции TF> (рисунок 4);
* <Описание переменных одного типа B> (рисунок 5);
* <Список переменных при описании L> (рисунок 6);
* <Блок O> (рисунок 7);
* <Операторы Z> (рисунок 8);
* <Один оператор Q> (рисунок 9);
* < Присваивание F> (рисунок 10);
* <Блок условия IF> (рисунок 11);
* <Вызов функции R> (рисунок 12);
* <Параметры функции P> (рисунок 13);
* <Список аргументов функции M> (рисунок 14);
* <Инициал I> (рисунок 15);
* <Оператор присваивания E> (рисунок 16);
* <Знак сравнения SC> (рисунок 17);
* <Знак операции SO> (рисунок 18);
* <Арифметические и логические операции A1, А2, А3, А4> (рисунок 19);

Ниже представлены диаграммы:

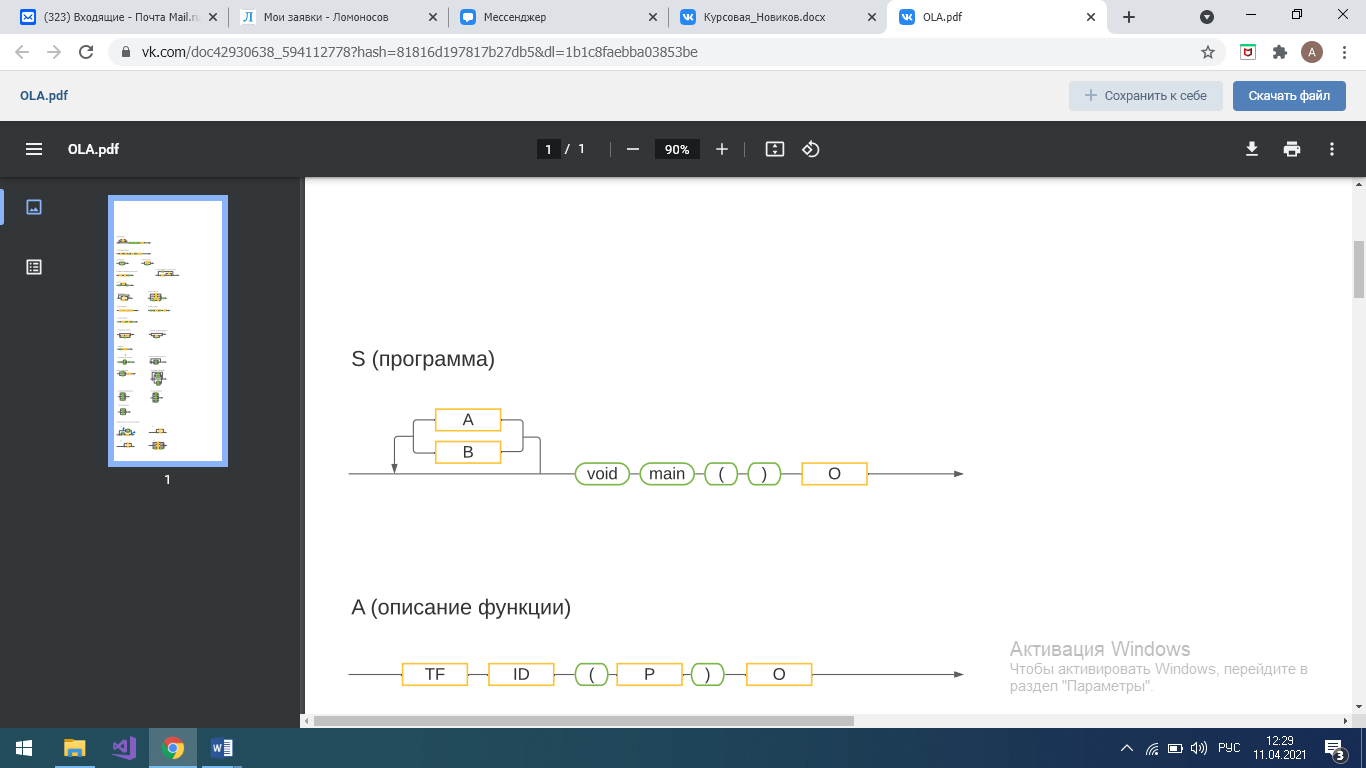


Рисунок 1 – Диаграмма программа

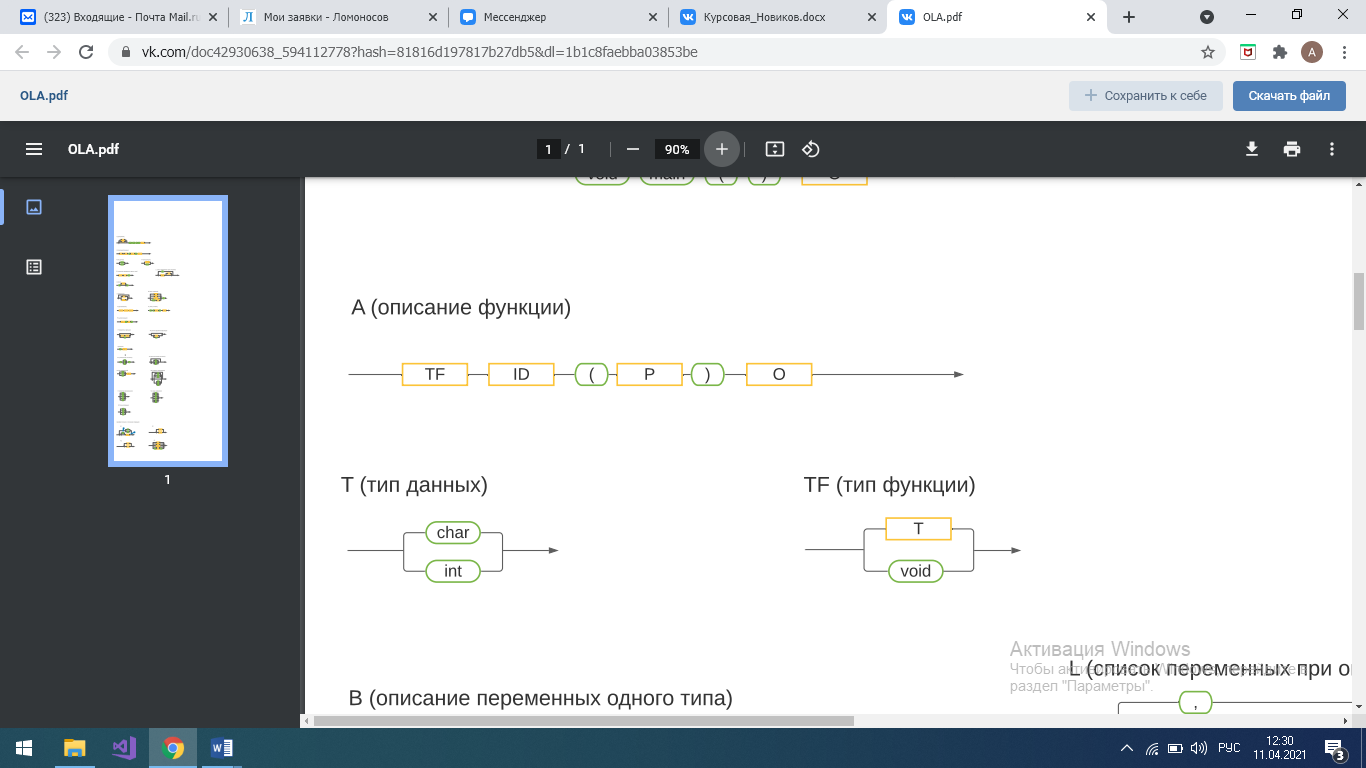


Рисунок 2 – Диаграмма описание функции

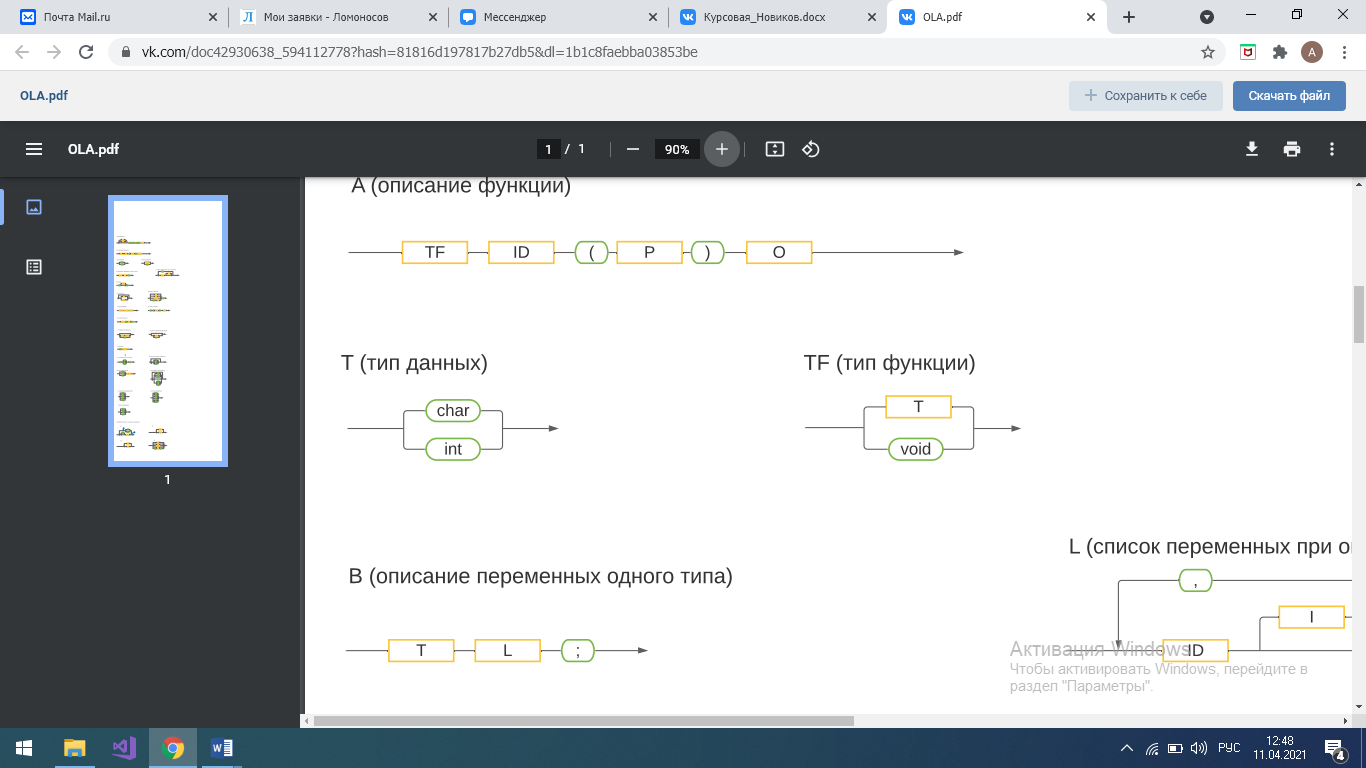


Рисунок 3 – Диаграмма тип данных

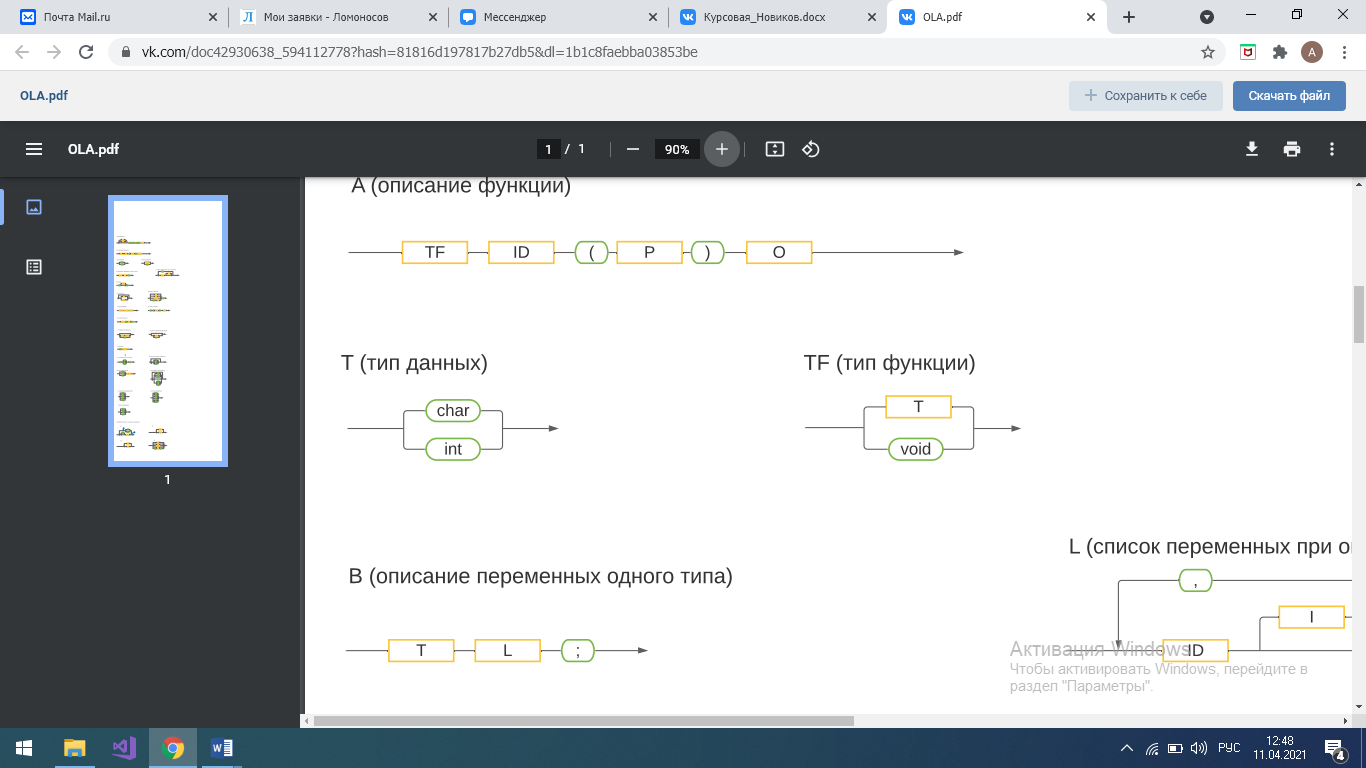


Рисунок 4 – Диаграмма тип функции

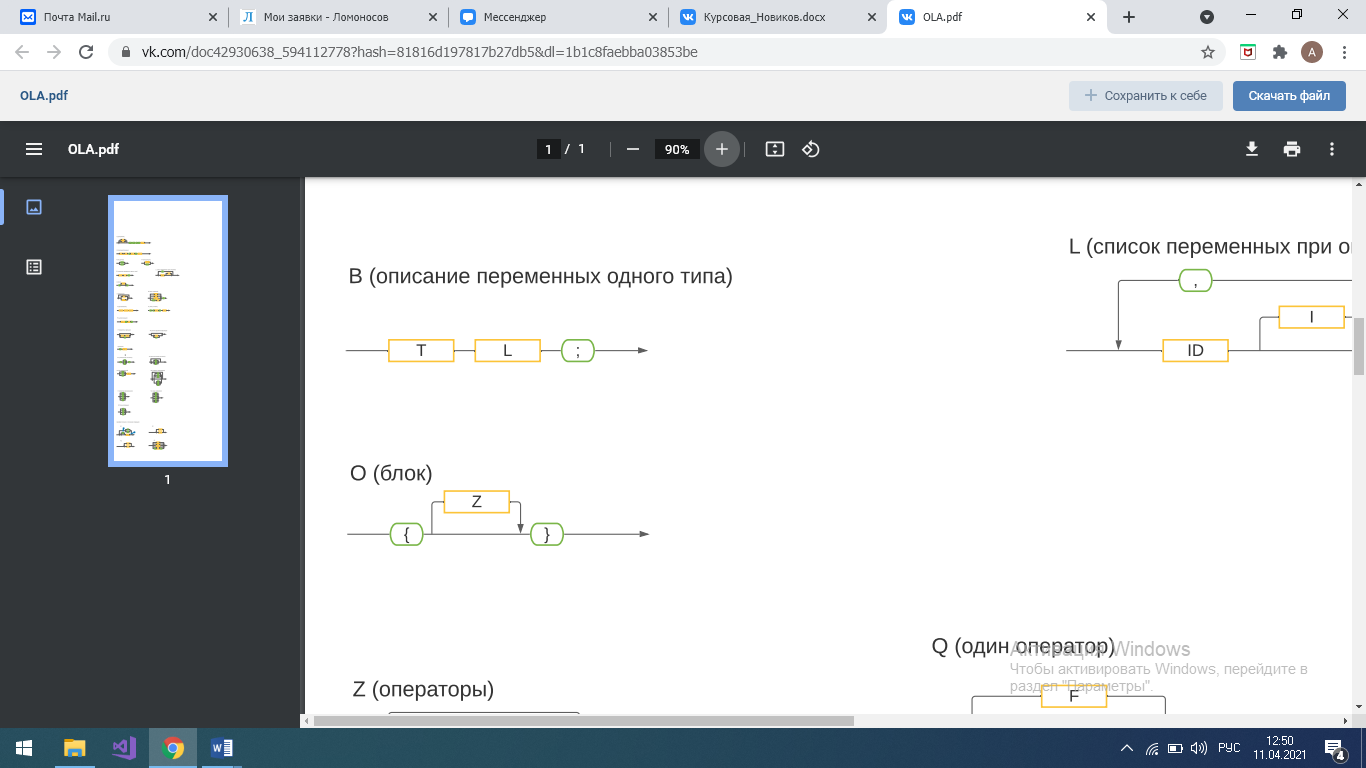


Рисунок 5 – Диаграмма описание переменных одного типа

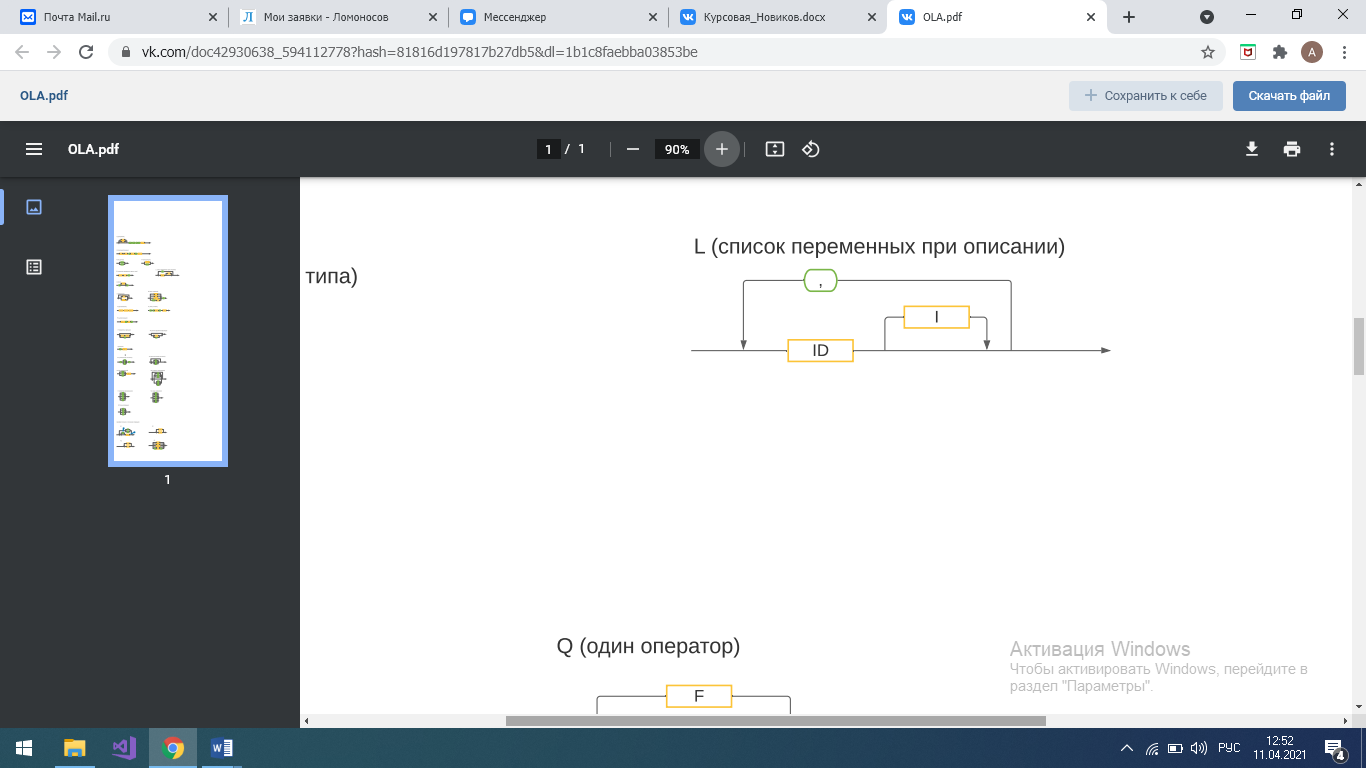


Рисунок 6 – Диаграмма список переменных при описании

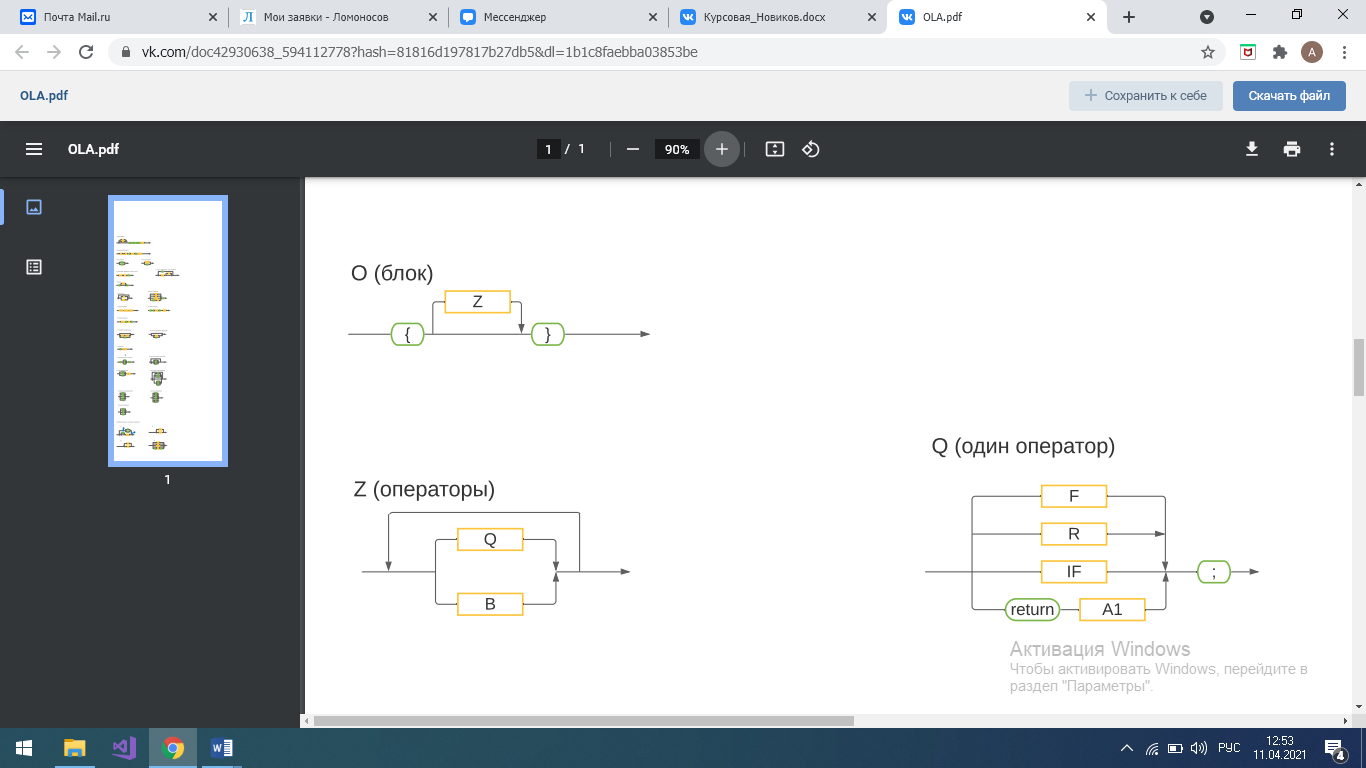


Рисунок 7 – Диаграмма блок

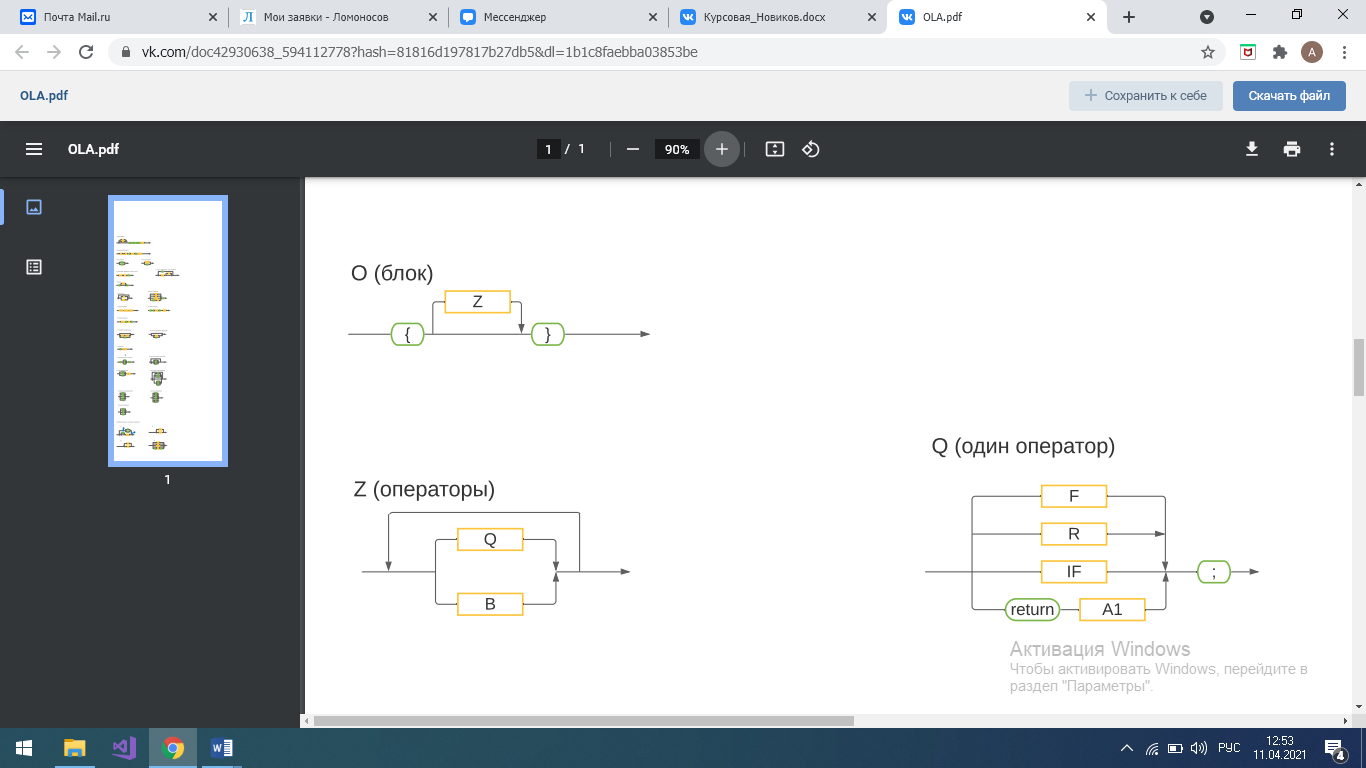


Рисунок 8 – Диаграмма операторы

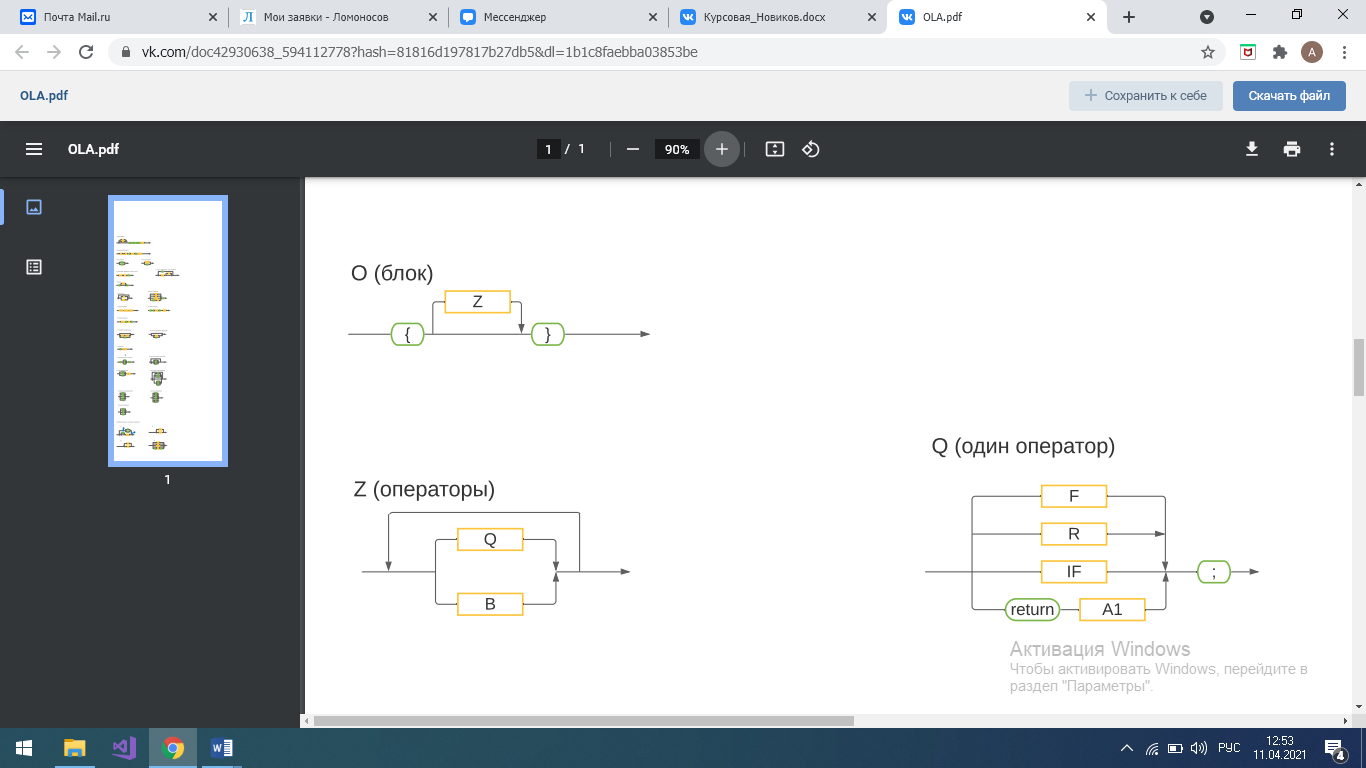


Рисунок 9 – Диаграмма один оператор

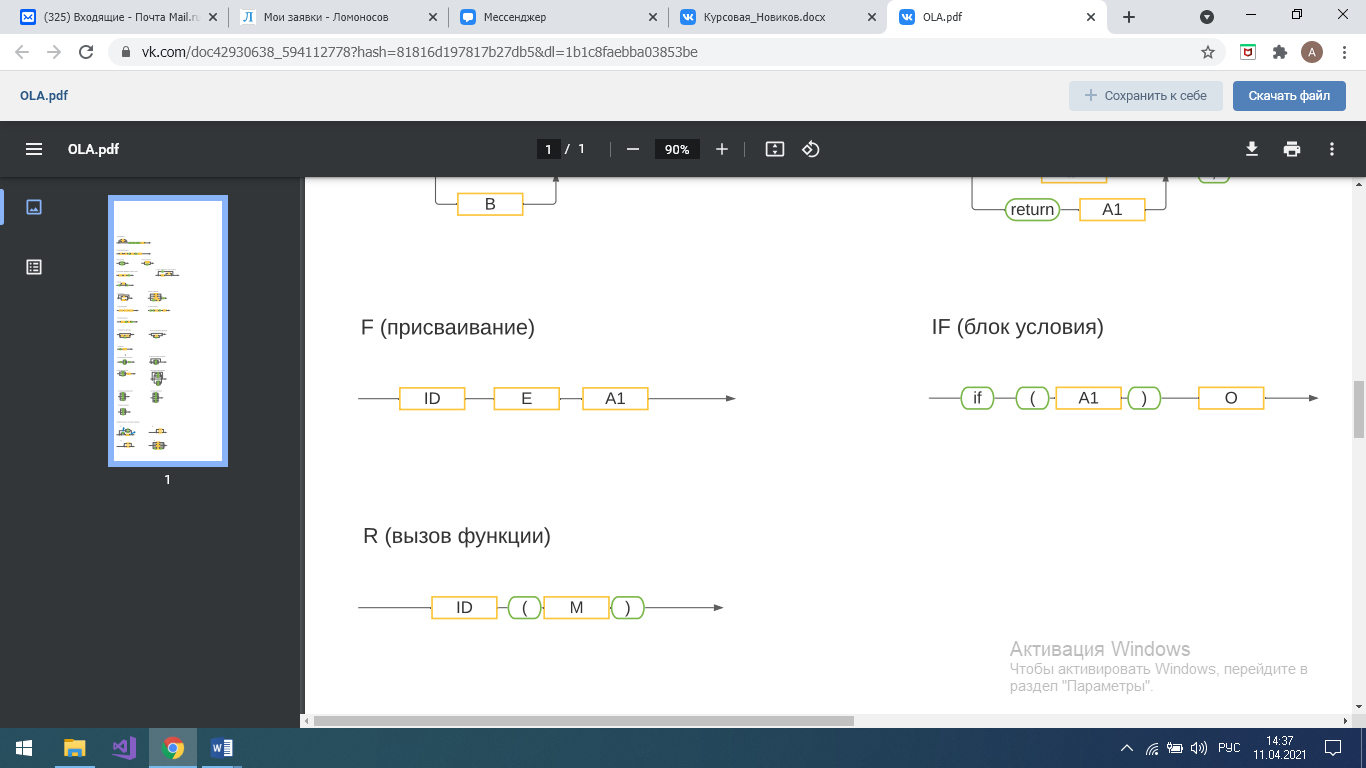


Рисунок 10 – Диаграмма присваивание

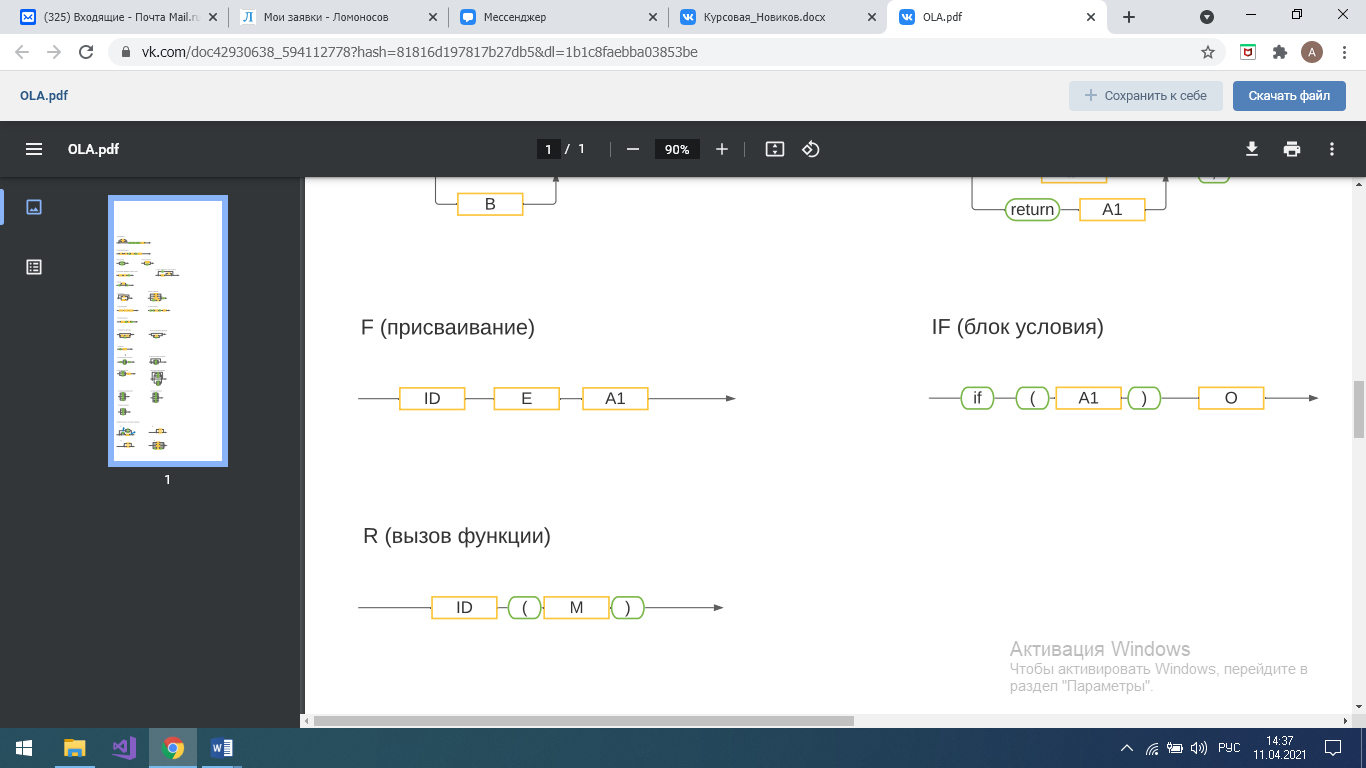


Рисунок 11 – Диаграмма блок условия

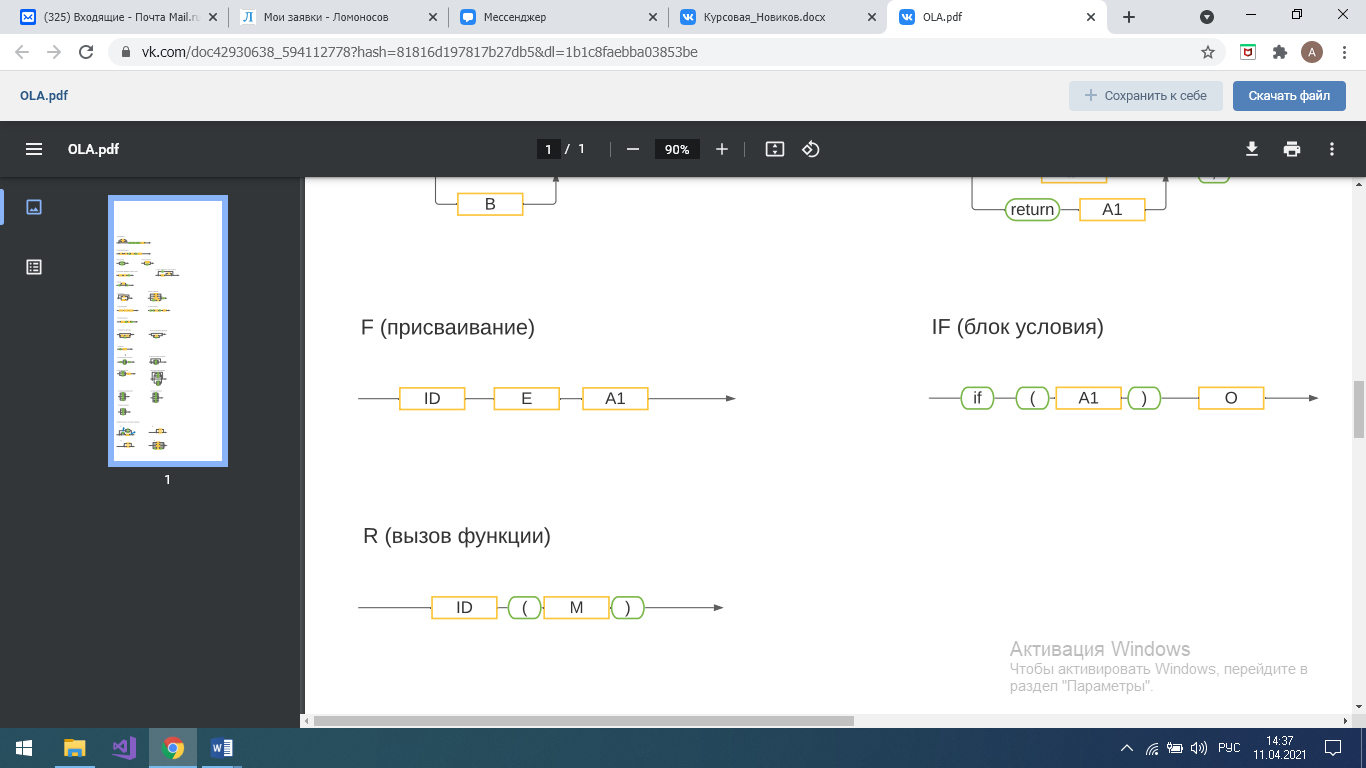


Рисунок 12 – Диаграмма вызов функции

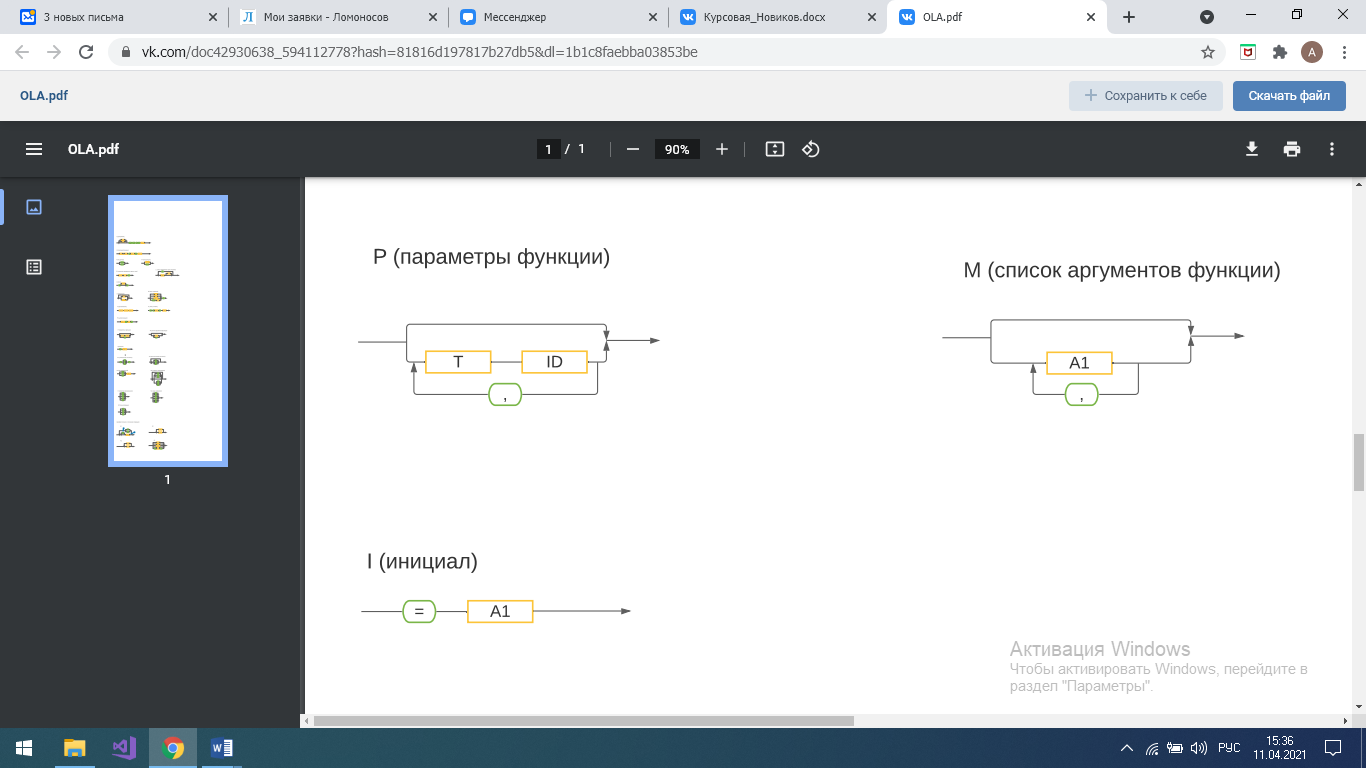


Рисунок 13 – Диаграмма парметры функции

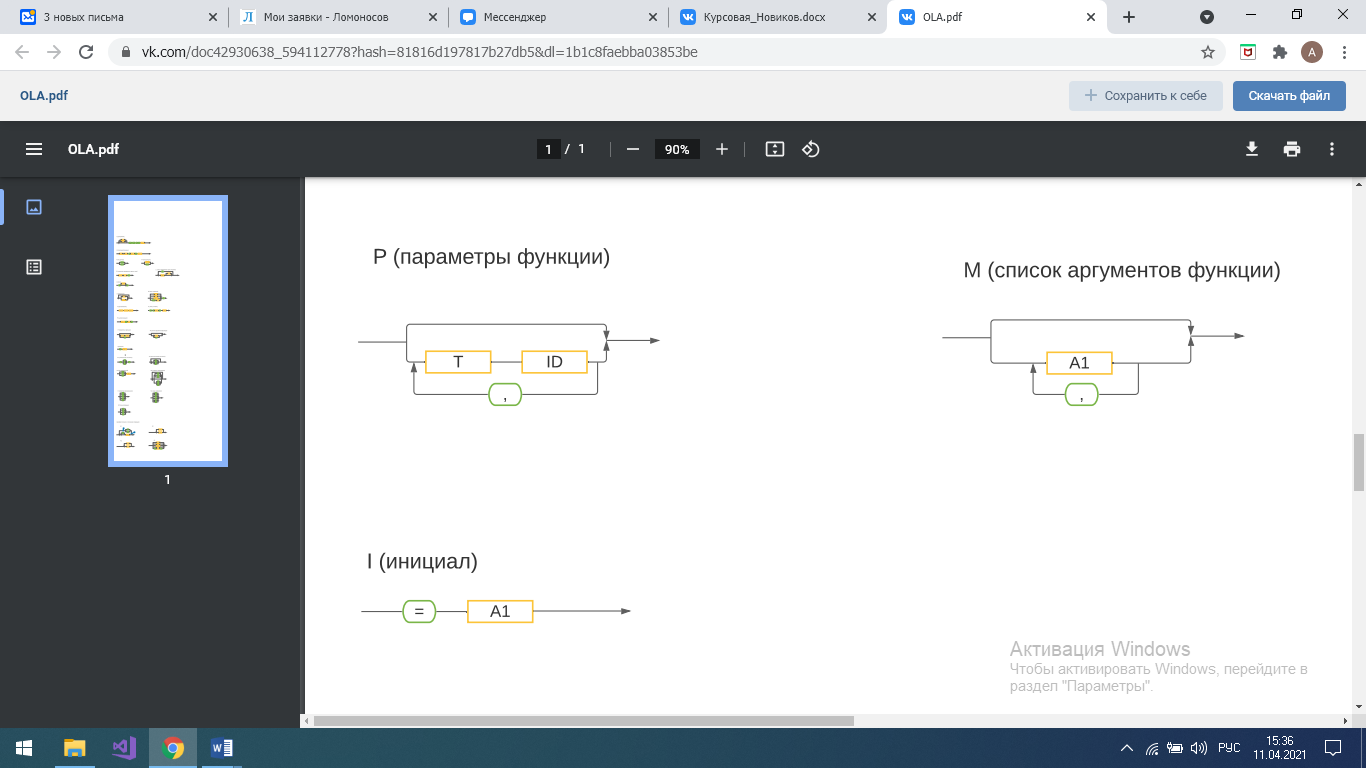


Рисунок 14 – Диаграмма список аргументов функции

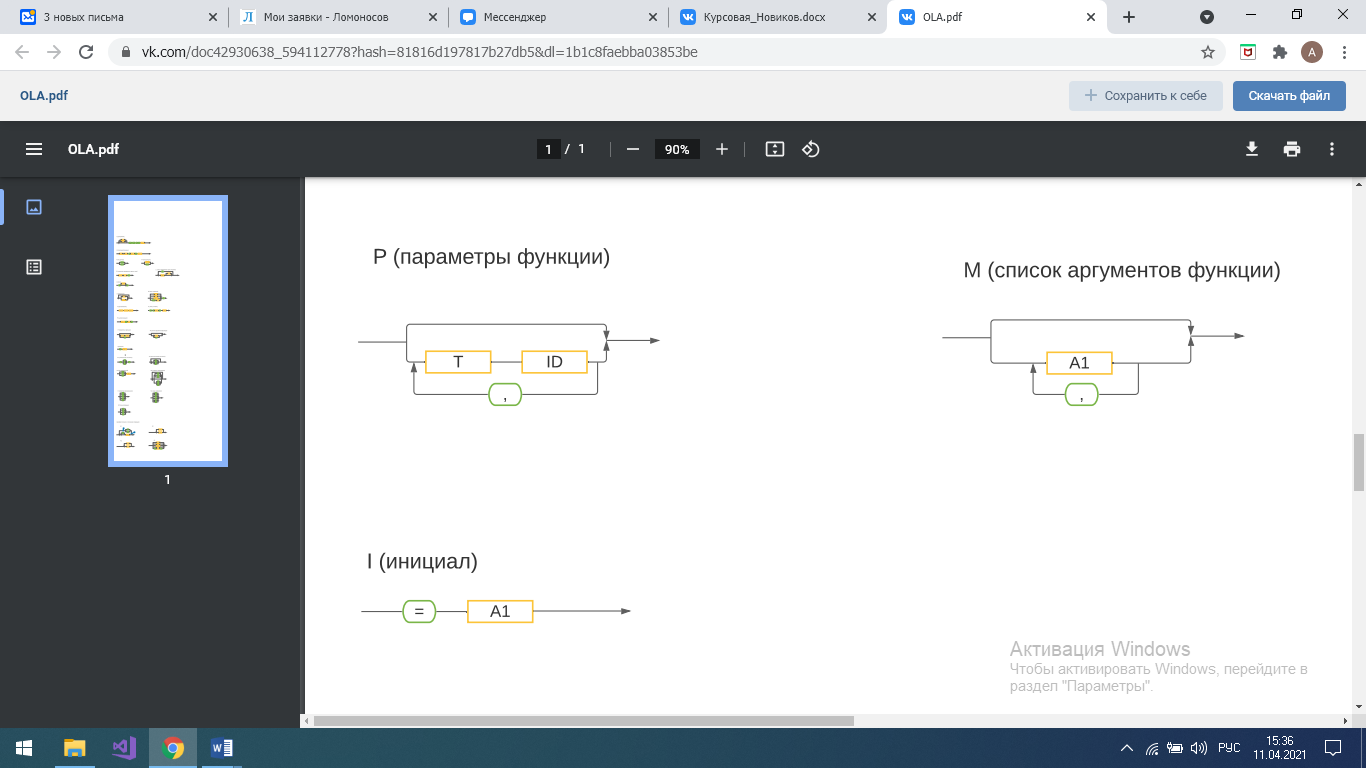


Рисунок 15 – Диаграмма инициал

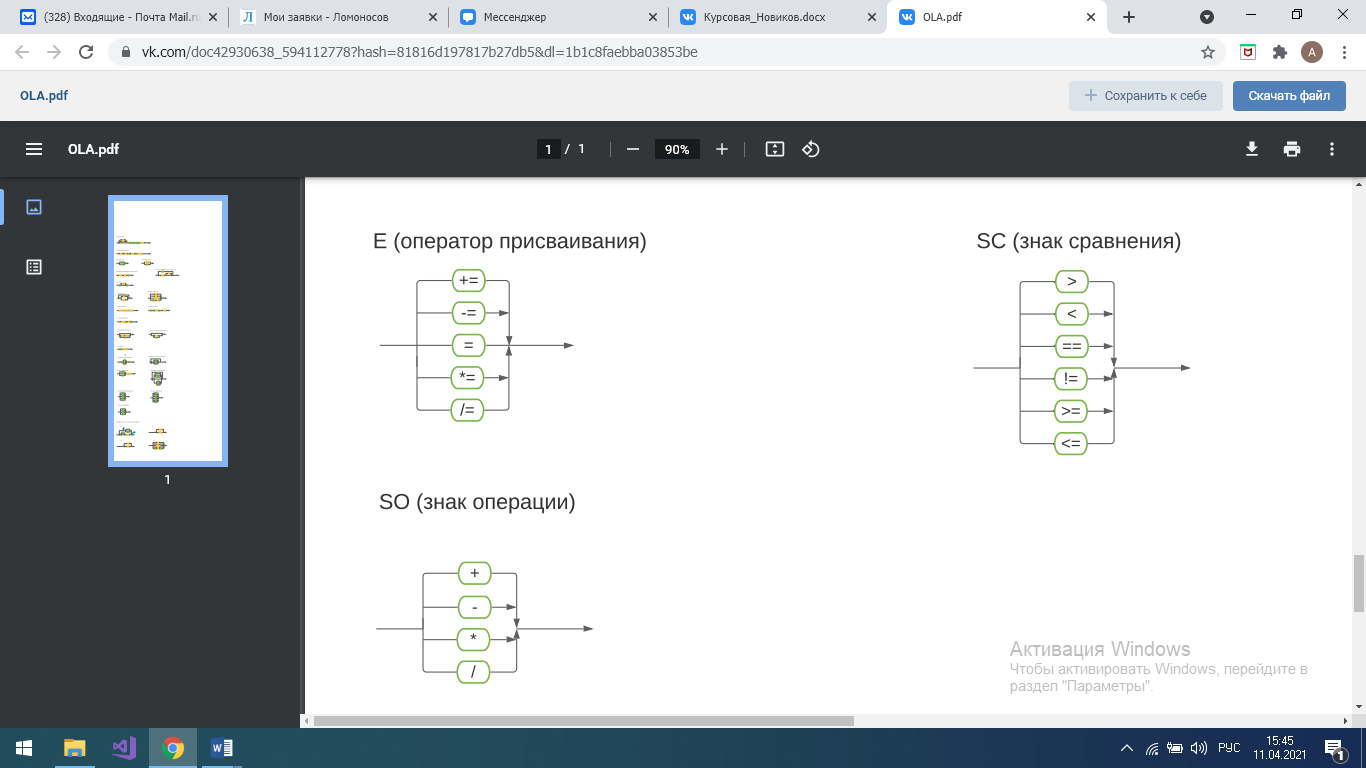


Рисунок 16 – Диаграмма оператор присваивания

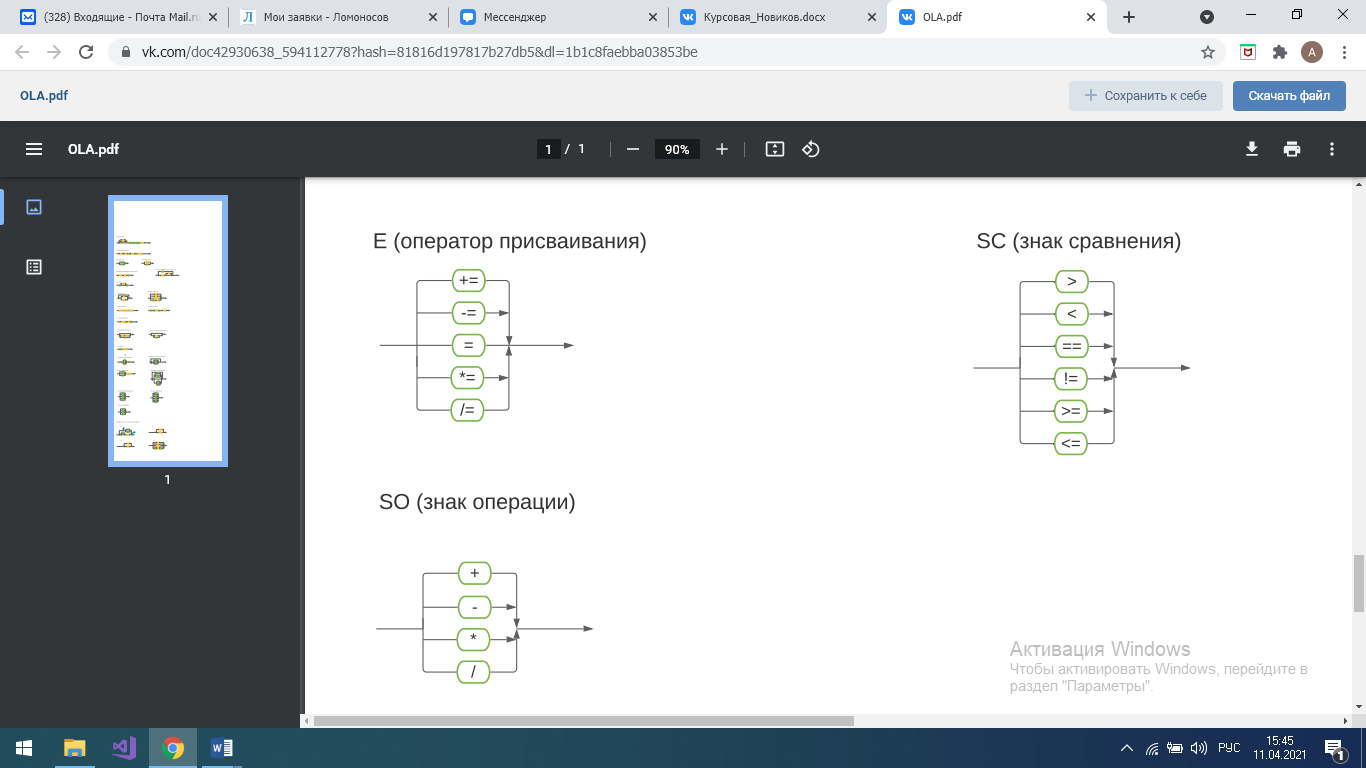


Рисунок 17 – Диаграмма знак сравнения

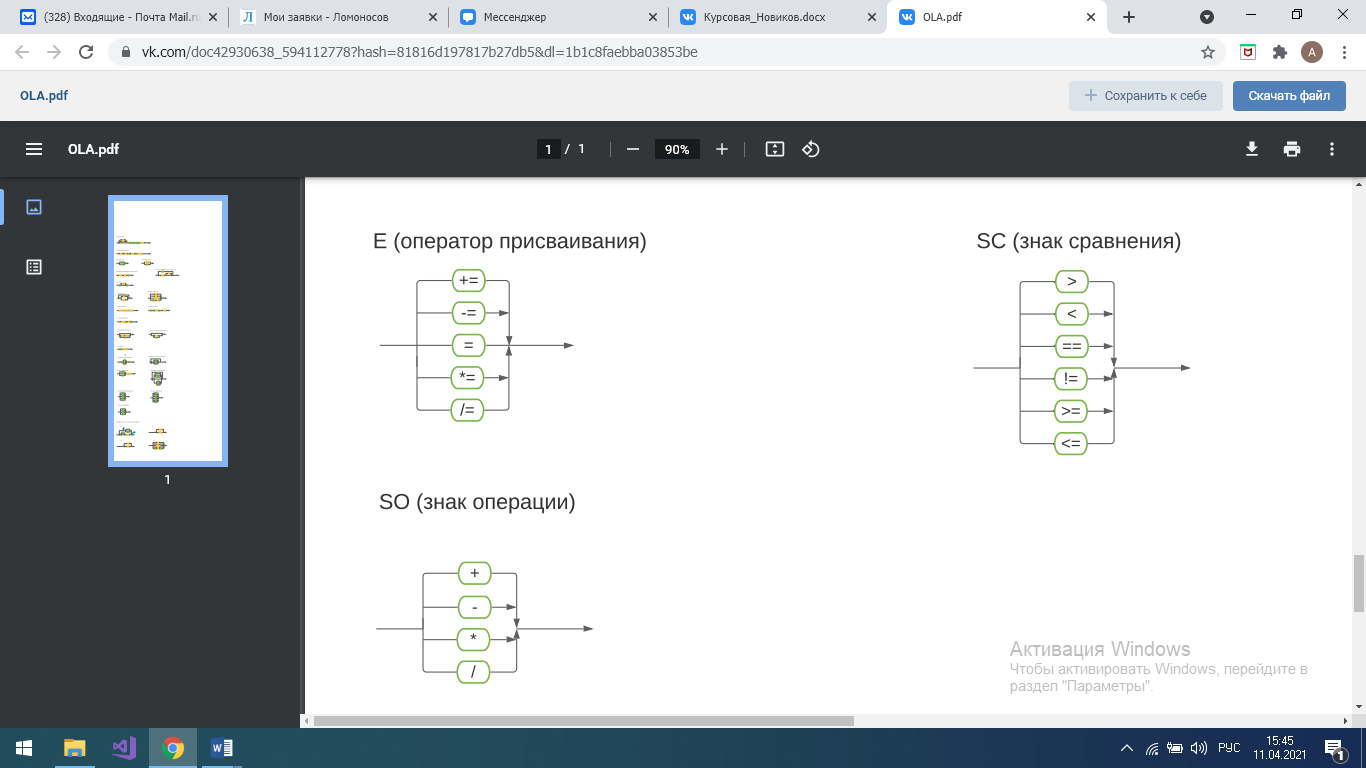


Рисунок 18 – Диаграмма знак операции

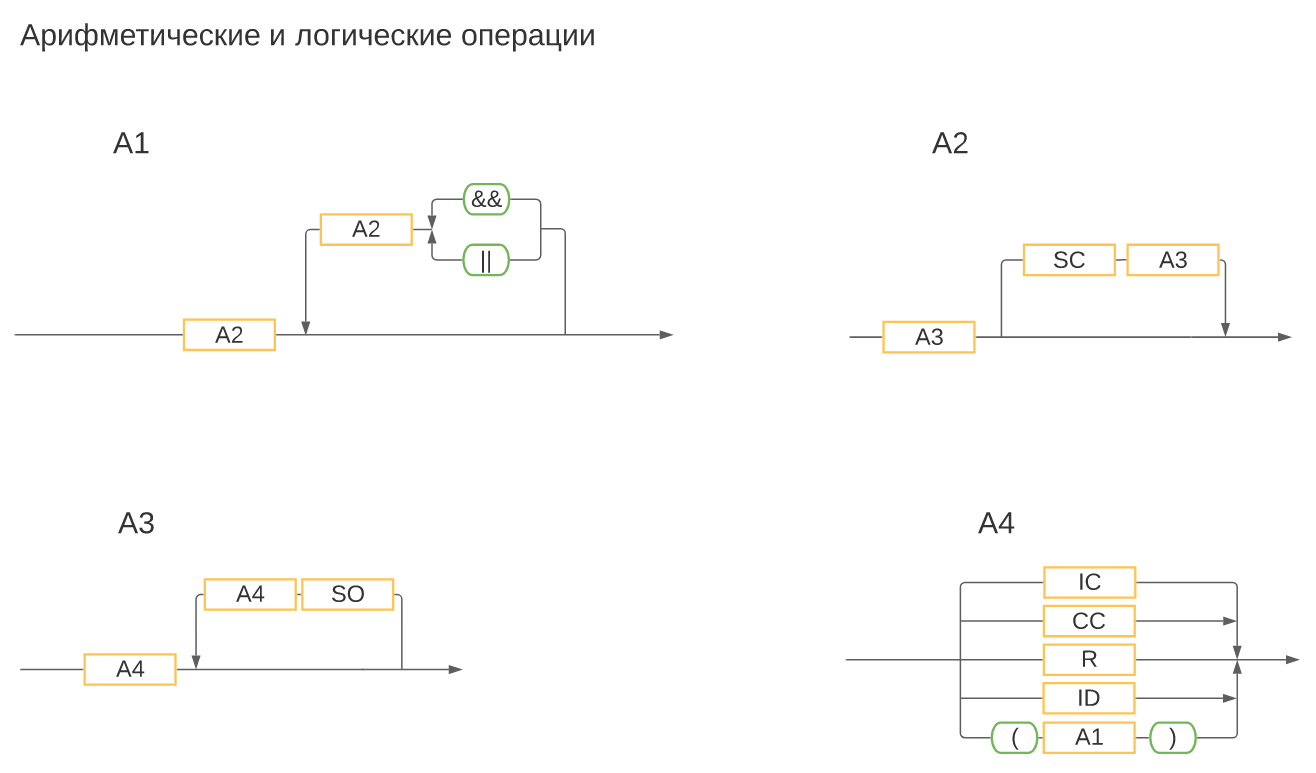


Рисунок 19 – Диаграмма арифметические и логические операции

В таблице 2 приведена таблица разметки (функции first, follow).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Функция** | **First 1** | **First 2** | **First 3** | **Follow** |
| <A> | char  int  void | идентификатор | ( | void |
| <B> | char  int | идентификатор | ; | void |
| <T> | char  int |  |  | идентификатор |
| <I> | = |  |  | ,  ; |
| <Z> | идентификатор  идентификатор  if  return  char  int |  |  | } |
| <Q> | идентификатор  идентификатор  if  return |  |  | идентификатор  идентификатор  if  return  char  int  } |
| <F> | идентификатор | +=  -=  =  \*=  /= |  | ; |
| <R> | идентификатор | ( |  | ; |
| <IF> | if |  |  | ; |
| <A1> | целочисл. конст.  симв.конст.  идентификатор  ( |  |  | ;  )  , |

# Программная реализация

Основной целью программы является разработка кроссплатформенного синтаксического анализатора для модельного языка с разветвляющимися конструкциями.

В процессе разработки анализатора модельного языка было написано консольное приложение. Для удобства командной разработки была использована система контроля версий – git. Основной удаленный репозиторий располагается на веб-сервисе для хостинга IT- проектов – GitHub.

Приложение состоит из 5 классов:

* Lexeme.java
* LexemeModel.java
* Scanner.java
* Analyzer.java
* Main.java

Класс Lexeme.java содержит перечисления названий лексем модельного языка и соответсвующие им коды.

Класс LexemeModel.java описывает лексему модельного языка.

Атрибуты класса LexemeModel.java:

* name – строкове представление лексемы;
* code – код лексемы.

За лексический анализ отвечает класс Scanner.java.

Атрибуты класса Scanner.java:

* MAXLEX – максимальный размер считываемой лексемы;
* stringKeywords – ключевые слова (идентификаторы);
* lexemeKeywords – коды ключевых слов;
* ignoreSymbols – массив игнорируемых символов;
* sourceText – текст программы для сканирования;
* textPointer – текущая позиция в тексте;
* lineNumber – текущий номер строки в тексте.

Методы класса Scanner.java:

* getNextLexeme() – считывает и возвращает очередную лексему;

Если метод вызывается первый раз, то происходит посимвольное считывание лексемы в локальную переменную, начиная с 0 элемента массива sourceText. Иначе считывание лексемы происходит с позиции конца последней считанной лексемы. В первую очередь, в цикле while пропускаются все игнорируемые символы (пробел, табуляция, перенос строки, комментарий) до встречи первой лексемы. Далее происходит идентификация лексемы по первому символу. Если однозначно лексему определить не удалось, то проверяется следующий символ до однозначной идентификации лексемы.

Функция не имеет типа возвращаемого значения.

* printError(String description, String lexeme) – выводит ошибки;

В качестве входных параметров функция получает два аргумена – строку с описанием ошибки и строку с названием лексемы.

Если аргумент с названием лексемы оказался пустым, то в консоль выводится только содержимое переменой описания ошибки.

В противном случае, в консоль выводятся описание ошибки, название лексемы с указанием того, что она является неверной, и номер строки с ошибкой.

После вывода сообщения об ошибке в консоль, программа завершается с кодом 1.

Функция не имеет типа возвращаемого значения.

* readProgramFromTextFile(String fileName) – считывает содержимое файла;

В данной функции единственным входным параметром является строка, содержащая путь к текстовому файлу, в котором хранится код программы на модельном языке.

Считывание файла происходит в цикле в объект класса StringBuilder с применением методов объектов классов File, Scanner. Затем содержимое объекта класса StringBuilder конвертируется в строку методом toString(), далее полученная строка разбивается на массив символов функцией toCharArray() и заносится в переменную класса Scanner – sourceText.

Функция не имеет типа возвращаемого значения.

* isEqualIgnoredSymbol(char lexemeSymb) – проверяет переданный символ на равенство с игнорируемыми;

Входным параметром функции является примитив типа char, содержащий в себе один считанный символ.

В цикле переменная типа char сравнивается с содержимым массива игнорируемых символов – атрибутом класса Scanner. Если входной параметр совпадает с одним из элементов массива игнорируемых символов, то функция возвращает true, в противном случае возвращается false.

* arrayToSting(char[] list) – преобразует массив символов в строку.

Данная функция является вспомогательным методом для конвертации массива символов в строку.

Функция имеет тип возвращаемого значения – String и возвращает полученную строку.

За синтаксический анализ отвечает класс Analyzer.java.

Анализатор осуществляет проверку считанной последовательности лексем на соответствие правилам модельного языка. Полноценная проверка последовательности лексем анализатором декомпозирована на отдельные блоки, представляющие собой функции без типа возвращаемого параметра, обрабатывающие отдельные конструкции программного кода. Данные конструкции представлены в диаграммах на рисунках 1 – 19.

Атрибуты класса Analyzer.java:

* scanner – объект класса Scanner.

Методы класса Analyzer.java:

* checkLexeme(Lexeme[] lexemes, String errorText) – проверяет очередные лексемы на равенство с переданным массивом лексем.

Данная функция не имеет типа возвращаемого значени, а в качестве входных параметров принимает массив лексем и текст ошибки.

В цикле происходит перебор массива лексем, и одновременное считывание очередной лексемы объектом scanner. В случае совпадения лексемы из массива с очередной лексемой цикл завершается и происходит выход из метода. Иначе – вызывается метод printError объекта scanner для вывода ошибки в консоль.

* setPointerTo(int pointer, int line) – устанавливает указатель объекта scanner.

Функция не имеет типа возвращаемого значения, а в качестве входных параметров принимает позицию указателя и номер строки.

Вызывается функция setTextPointer объекта scanner, и в качестве параметра передаётся позиция указателя, а затем функция setLineNumber с номером строки в качестве параметра.

* S() – проверка всей программы на модельном языке (далее – это имеется в виду);

Данная функция не имеет входных параметров и типа возвращаемого значения (все функции проверки конструкций модельного языка не имеют входных параметров и типа возвращаемого значения). Функция S вызывает метод класса Scanner getNextLexeme() для получения очередной лексемы. Далее запускается цикл while, выполнняемый до тех пор, пока очередная лексема не будет являться лексемой, обозначающей конец программы. Внутри цикла происходит определение, проверка какой именно конструкции модельного языка должна быть вызвана.

while (lexemeType.getCode() != Lexeme.*T\_END*.lexemeCode) {  
 if (lexemeType.getCode() == Lexeme.*T\_CHAR*.lexemeCode || lexemeType.getCode() == Lexeme.*T\_INT*.lexemeCode) {  
 ...  
 lexemeType = scanner.getNextLexeme();  
  
 if (lexemeType.getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 ...  
 A(); *// Проверка описания функции* } else {  
 ...  
 B(); *// Проверка описания переменных одного типа* }  
  
 } else if (lexemeType.getCode() == Lexeme.*T\_VOID*.lexemeCode) {  
 lexemeType = scanner.getNextLexeme();  
 if (lexemeType.getCode() == Lexeme.*T\_MAIN*.lexemeCode) {  
 *// Проверка функции main* ...  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 O();  
 } else {  
 ...  
 A(); *// Проверка описания функции*  
 }  
 }  
 ...  
}

* A() – проверка конструкции описания функции модельного языка (далее – это имеется в виду);

В методе сначала происходит вызов функций для проверки конструкций модельного языка, описывающей тип функции, имени переменной, параметров функции и её тела. А также проверяется наличие открывающейся и закрывающейся круглых скобок.

* T() – проверка конструкции, описывающей типы данных;

Очередная лексема проверяется на соответсвие лексеме, обозначающей тип данных – char или int.

* TF() – проверка конструкции, описывающей типы функции;

Очередная лексема проверяется на соответсвие лексеме, обозначающей тип данных функции – char, int или void.

* B() – проверка конструкции описания переменных одного типа;

В методе сначала происходит вызов функций для проверки конструкций модельного языка, описывающей тип данных и списка переменных при описании. В конце метода проверяется наличие знака точки с запятой.

* L() – проверка конструкции списка переменных при описании;
* O() – проверка конструкции блока { };

В методе проверяется сначала наличие открывающейся фигурной скобики. Если она есть и следующая лексема – незакрывающаяся фигурная скобка, то вызывается Z(). Далее проверяется наличие закрывающейся фигурной скобки, если ее нет, то выводится сообщение об ошибке.

* Z() – проверка конструкции операторов;

В методе происходит последовательное считывание в цикле лексем до правой закрывающейся скобки. Если считанная лексема совпадает с типом char или int, вызывает В(), иначе Q().

do {  
 setPointerTo(lastPointer, lastLine);  
 lexemeType = scanner.getNextLexeme();  
 setPointerTo(lastPointer, lastLine);  
 if (lexemeType.getCode() == Lexeme.*T\_CHAR*.lexemeCode || lexemeType.getCode() == Lexeme.*T\_INT*.lexemeCode) {  
 B();  
 } else {  
 Q();  
 }  
  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
} while (scanner.getNextLexeme().getCode() != Lexeme.*T\_RIGHT\_BRACE*.lexemeCode);

* Q() – проверка одного оператора;

В методе осуществляется проверка на наличие ключевого слова if.

Если условие выполнилось, то происходит вызов функции IF() для проверки конструкции блока if.

В противном случае, метод вызывает проверяет очередную лексему на равенство ключевому слову return. В случае совпадения вызывается функция A1(). Иначе – происходит проверка очередной лексемы на совпадение с открывающейся круглой скобкой. При выполнении условия это означает, что идентифицирована конструкция вызова функции и вызывается метод R(). В другом случае – идентифицирована конструкция присваивания и вызывается метод F();

* F() – проверка присваивания;

Сначала в методе очередная лексема проверяется на соответсвие лексеме, обозначающей идентификатор (имя переменной или функции). Затем – вызов функций для проверки наличия операции присваивания.

* IF() – проверка блока условий;

В начале метода осуществляется проверка лексемы на наличие ключевого слова if. Затем функция определяет присутствие круглых скобок и операции (логических и/или арифметических) внутри. Для проверки операции вызывается функция А1(). Далее вызывается метод О() для проверки блока внутри условного оператора.

*// if*checkLexeme(Lexeme.*T\_IF*, "Ожидалось ключевое слово if");  
checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
A1();  
checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
O();

Помимо этого, в методе реализована проверка конструкций else if, else.

*// else if*int lastPointer = scanner.getTextPointer();  
int lastLine = scanner.getLineNumber();  
while (scanner.getNextLexeme().getCode() == Lexeme.*T\_ELSE*.lexemeCode && scanner.getNextLexeme().getCode() == Lexeme.*T\_IF*.lexemeCode) {  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 A1();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 O();  
 lastLine = scanner.getLineNumber();  
 lastPointer = scanner.getTextPointer();  
}  
setPointerTo(lastPointer, lastLine);  
  
  
*// else*lastPointer = scanner.getTextPointer();  
lastLine = scanner.getLineNumber();  
if (scanner.getNextLexeme().getCode() == Lexeme.*T\_ELSE*.lexemeCode) {  
 O();  
} else {  
 setPointerTo(lastPointer, lastLine);  
}

* R() – проверка вызова функции;

В методе происходит проверка на наличие открывающихся и закрывающихся круглых скобок, а также вызывается метод М() (проверка списка аргументов функции).

* P() – проверка параметров функции;

Метод с помощью цикла do while осуществляет последовательное считывание параметров функции.

if (scanner.getNextLexeme().getCode() != Lexeme.*T\_RIGHT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 do {  
 T();  
 checkLexeme(Lexeme.*T\_ID*, "Ожидалось имя перемнной или функции");  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 } while (scanner.getNextLexeme().getCode() == Lexeme.*T\_COM*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
}  
setPointerTo(lastPointer, lastLine);

* M() – проверка списка аргументов функции;

В методе происходит проверка на наличие открывающихся и закрывающихся круглых скобок, а также вызывается метод А1() (арифметические и/или логические операции).

* I() – проверка инициализации;

Метод осуществляет проверку на наличие символа '=' и вызывает метод А1() (арифметические и/или логические операции).

* E() – проверка оператора присваивания;

Метод осуществляет проверку на наличие одного из символов операторов присваивания.

* A1() – проверка логических операций;

Считывание лексем и их сравнение происходит в циклах. Внешний цикл осуществляет проверку на наличие следующих условий. Внутренний цикл проверяет отрицание логического выражения.

do {  
 do {  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 }  
 while (scanner.getNextLexeme().getCode() == Lexeme.*T\_NOT*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 A2();  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
} while (scanner.getNextLexeme().getCode() == Lexeme.*T\_AND*.lexemeCode || scanner.getNextLexeme().getCode() == Lexeme.*T\_OR*.lexemeCode);  
setPointerTo(lastPointer, lastLine);

* A2() – проверка операций сравнения;

В методе изначально вызывается А3(). Затем, при истинности одного из условий сравнения на лексему оператора сравнения, А3() вызывается повторно.

if (lexemeType.getCode() == Lexeme.*T\_MORE*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_LESS*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_MORE\_AND\_EQUAL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_LESS\_AND\_EQUAL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_EQUAL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_NOT\_EQUAL*.lexemeCode) {  
 A3();  
} else {  
 setPointerTo(lastPointer, lastLine);  
}

* A3() – проверка арифметических операций;

В методе изначально вызывается А4(). Затем, при истинности одного из условий сравнения на лексему арифметического оператора, А4() вызывается повторно.

* A4() – проверка значений (переменной, константы, результата вызова функции, выражения).

В методе осуществляется проверка на наличие открывающейся круглой скобки.

Если условие выполнилось, то происходит вызов А1().

В противном случае, метод вызывает R(), если лексема не совпала с именем переменной, выдается соответствующее сообщение об ошибке.

if (lexemeType.getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 A1();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
} else if (lexemeType.getCode() == Lexeme.*T\_ID*.lexemeCode) {  
 if (scanner.getNextLexeme().getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 R();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 checkLexeme(Lexeme.*T\_ID*, "Ожидалось имя перемнной или функции");  
 }  
} else {  
 setPointerTo(lastPointer, lastLine);  
 Lexeme[] lexemes = new Lexeme[]{Lexeme.*T\_CONST\_CHAR*, Lexeme.*T\_CONST\_INT*};  
 checkLexeme(lexemes, "Ожидалась целочисленная или символьная константа");  
}

# Тестирование программы

Для проверки корректности работы синтаксического анализатора проведем тестирование на следующих вариантах листинга программы, написанной на модельном языке:

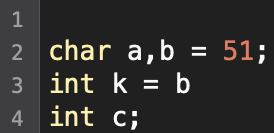


Рисунок 20 – Листинг кода с ошибкой

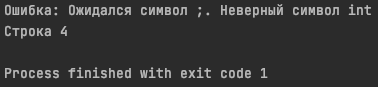


Рисунок 21 – Сообщение об ошибке

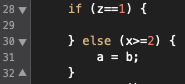


Рисунок 22 – Листинг кода с ошибкой

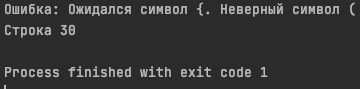


Рисунок 23 – Сообщение об ошибке

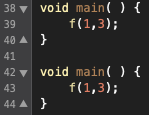


Рисунок 24 – Листинг кода с ошибкой

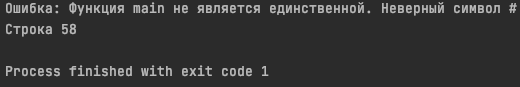


Рисунок 25 – Сообщение об ошибке

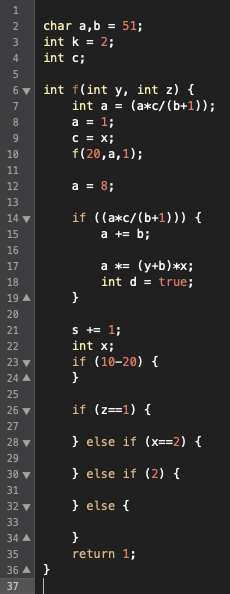


Рисунок 26 – Листинг кода с ошибкой

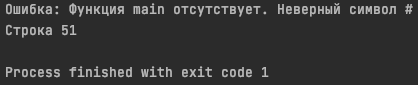


Рисунок 27 – Сообщение об ошибке

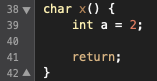


Рисунок 28 – Листинг кода с ошибкой

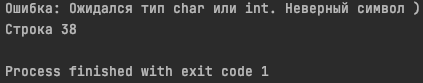


Рисунок 29 – Сообщение об ошибке



Рисунок 30 – Листинг кода с ошибкой

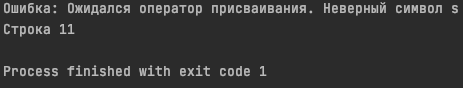


Рисунок 31 – Сообщение об ошибке



Рисунок 32 – Листинг кода с ошибкой

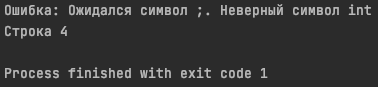


Рисунок 33 – Сообщение об ошибке

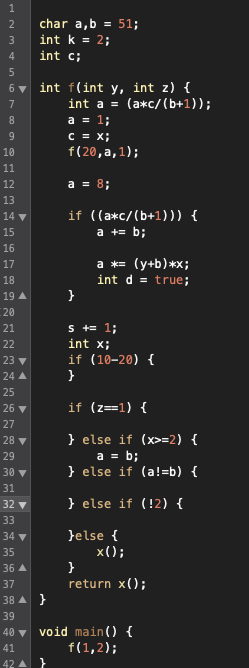


Рисунок 34 – Листинг корректного кода



Рисунок 35 – Сообщение об успешной проверке

# Заключение

В данной работе было описано создание кроссплатформенного синтаксического анализатора для модельного языка с разветвляющимися конструкциями. Были спроектированы конструкции языка и написан по ним программный код, а также обеспечен вывод подробной информации о процессе.

# Cписок использованных источников

1. Крючкова Е. Н. Основы теории языков программирования и методы трансляции. [Электронный ресурс]: Учебное пособие. — Электрон. дан. — Барнаул: АлтГТУ, 2010. — Режим доступа: http://elib.altstu.ru/eum/download/pm/kruchkova\_teorjaz.pdf
2. Вирт, Н. Построение компиляторов [Электронный ресурс] / Н. Вирт. — Электрон. дан. — Москва : ДМК Пресс, 2010. — 192 с. — Режим доступа: https://e.lanbook.com/book/1262. — Загл. с экрана.
3. Карпов Ю.Г. Теория и технология программирования. Основы построения трансляторов. — СПб.: БХВ-Петербург, 2005. — 272 с.

# Приложение А. Исходный код программы

Lexeme.java

packagecom.linguisticanalysis.scanner.enums;  
  
*/\*\*  
 \* Лексема  
 \*/*package com.linguisticanalysis.enums;  
  
*/\*\*  
 \* Лексема  
 \*/*public enum Lexeme {  
 *T\_ID*(1),  
 *T\_INT*(10),  
 *T\_CHAR*(11),  
 *T\_VOID*(12),  
 *T\_MAIN*(13),  
 *T\_RETURN*(14),  
 *T\_IF*(15),  
 *T\_ELSE*(16),  
  
 *// Знаки операций  
 T\_EQ*(20),  
 *T\_PLUS*(21),  
 *T\_SUB*(22),  
 *T\_MUL*(23),  
 *T\_DIV*(24),  
 *T\_PLUS\_EQ*(25),  
 *T\_SUB\_EQ*(26),  
 *T\_MUL\_EQ*(27),  
 *T\_DIV\_EQ*(28),  
 *T\_NOT*(29),  
 *T\_AND*(30),  
 *T\_OR*(31),  
  
 *// Знаки сравнения  
 T\_MORE*(35),  
 *T\_LESS*(36),  
 *T\_EQUAL*(37),  
 *T\_NOT\_EQUAL*(38),  
 *T\_MORE\_AND\_EQUAL*(39),  
 *T\_LESS\_AND\_EQUAL*(40),  
  
 *T\_CONST\_INT*(45),  
 *T\_CONST\_CHAR*(46),  
 *T\_SEMI*(47),  
 *T\_COM*(48),  
 *T\_LEFT\_ROUND\_BR*(49),  
 *T\_RIGHT\_ROUND\_BR*(50),  
 *T\_LEFT\_BRACE*(51),  
 *T\_RIGHT\_BRACE*(52),  
  
 *T\_END*(1000),  
 *T\_ERROR*(2000);  
  
  
 public final int lexemeCode;  
  
 private Lexeme(int lexemeCode) {  
 this.lexemeCode = lexemeCode;  
 }  
}

LexemeModel.java

package com.linguisticanalysis.analyzer;  
  
public class LexemeModel {  
 private String name;  
 private int code;  
  
 public LexemeModel(String name, int code) {  
 this.name = name;  
 this.code = code;  
 }  
  
 public String getName() {  
 return name;  
 }  
 public int getCode() {  
 return code;  
 }  
}

Scanner.java

package com.linguisticanalysis.scanner;  
import com.linguisticanalysis.analyzer.LexemeModel;  
import com.linguisticanalysis.enums.Lexeme;  
import java.io.File;  
import java.io.IOException;  
  
public class Scanner {  
  
 */\*\*  
 \* Максимальный размер лексемы  
 \*/* private static final int *MAXLEX* = 20;  
 */\*\*  
 \* Ключевые слова (идентификаторы)  
 \*/* private static final String[] *stringKeywords* = {  
 "int",  
 "char",  
 "void",  
 "main",  
 "return",  
 "if",  
 "else"  
 };  
  
 */\*\*  
 \* Ключевые слова (коды идентификаторов)  
 \*/* private static final Lexeme[] *lexemeKeywords* = {  
 Lexeme.*T\_INT*,  
 Lexeme.*T\_CHAR*,  
 Lexeme.*T\_VOID*,  
 Lexeme.*T\_MAIN*,  
 Lexeme.*T\_RETURN*,  
 Lexeme.*T\_IF*,  
 Lexeme.*T\_ELSE* };  
  
 */\*\*  
 \* Игнорируемые символы  
 \*/* private static final char[] *ignoreSymbols* = {  
 ' ',  
 '\t',  
 '\n'  
 };  
  
 */\*\*  
 \* Текст программы для сканирования  
 \*/* private char[] sourceText;  
  
 */\*\*  
 \* Текущая позиция в тексте  
 \*/* private int textPointer;  
  
 */\*\*  
 \* Текущий номер строки в тексте  
 \*/* private int lineNumber;  
  
 public Scanner(String filePathWithName) {  
 readProgramFromTextFile(filePathWithName);  
 textPointer = 0;  
 lineNumber = 0;  
 }  
  
 */\*\*  
 \* Получение очередной лексемы  
 \*/* public LexemeModel getNextLexeme() {  
 int i = 0; *// лексема заполняется с позиции i* char[] currentLexemeSymbols = new char[*MAXLEX*];  
  
 *// Пропускаем все пробелы, табуляции, пустые строки, комментарии* while (isEqualIgnoredSymbol(sourceText[textPointer]) || (sourceText[textPointer] == '/' && sourceText[textPointer + 1] == '/')) {  
  
 *// пробел, табуляция* if ((sourceText[textPointer] == ' ') || (sourceText[textPointer] == '\t'))  
 textPointer++;  
  
 *// пустая строка* else if (sourceText[textPointer] == '\n') {  
 lineNumber++;  
 textPointer++;  
 }  
  
 *// комментарии* else if (sourceText[textPointer] == '/' && sourceText[textPointer + 1] == '/') {  
 textPointer += 2;  
  
 *// пока не новая строка или не конец программы - пропускаем* while (sourceText[textPointer] != '\n' && sourceText[textPointer] != '\0') {  
 textPointer++;  
 }  
 }  
 }  
  
 *// Проверка на конец программы* if (sourceText[textPointer] == '\0') {  
 currentLexemeSymbols[0] = '#';  
  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_END*.lexemeCode);  
 }  
  
 *// id или keyword* if (((sourceText[textPointer] >= 'a') && (sourceText[textPointer] <= 'z')) ||  
 ((sourceText[textPointer] >= 'A') && (sourceText[textPointer] <= 'Z'))) {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 while (((sourceText[textPointer] >= 'a') && (sourceText[textPointer] <= 'z')) ||  
 ((sourceText[textPointer] >= 'A') && (sourceText[textPointer] <= 'Z')) ||  
 ((sourceText[textPointer] >= '0') && (sourceText[textPointer] <= '9'))) {  
 if (i < *MAXLEX* - 1) {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
 } else {  
 textPointer++;  
 }  
 }  
  
 *// сверяем с ключевыми словами* for (int j = 0; j < *lexemeKeywords*.length; j++) {  
 if (arrayToSting(currentLexemeSymbols).equals(*stringKeywords*[j])) {  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), *lexemeKeywords*[j].lexemeCode);  
 }  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_ID*.lexemeCode);  
 }  
  
 *// целочисленная константа* else if ((sourceText[textPointer] >= '0') &&  
 (sourceText[textPointer] <= '9')) {  
  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 boolean errorLengthFlag = false;  
  
 *//ошибка длины константы* while ((sourceText[textPointer] >= '0') &&  
 (sourceText[textPointer] <= '9')) {  
  
 if (i < *MAXLEX* - 1) {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
 } else {  
 errorLengthFlag = true;  
 textPointer++;  
 }  
 }  
  
 if ((sourceText[textPointer] >= 'a' && sourceText[textPointer] <= 'z') || (sourceText[textPointer] >= 'A' && sourceText[textPointer] <= 'Z')) {  
 while (!isEqualIgnoredSymbol(sourceText[textPointer]) &&  
 sourceText[textPointer] != ';') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
 }  
 printError("", arrayToSting(currentLexemeSymbols));  
 }  
  
 if (errorLengthFlag) {  
 printError("Слишком длинная целочисленная константа", arrayToSting(currentLexemeSymbols));  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_ERROR*.lexemeCode);  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_CONST\_INT*.lexemeCode);  
 }  
  
 *// \* \*=* else if (sourceText[textPointer] == '\*') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_MUL\_EQ*.lexemeCode);  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_MUL*.lexemeCode);  
 }  
  
 *// / /=* else if (sourceText[textPointer] == '/') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_DIV\_EQ*.lexemeCode);  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_DIV*.lexemeCode);  
 }  
  
 *// + +=* else if (sourceText[textPointer] == '+') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_PLUS\_EQ*.lexemeCode);  
 }  
  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_PLUS*.lexemeCode);  
 }  
  
 *// - -=* else if (sourceText[textPointer] == '-') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_SUB*.lexemeCode);  
 }  
  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_SUB*.lexemeCode);  
 }  
  
  
 *// &&* else if (sourceText[textPointer] == '&') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '&') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_AND*.lexemeCode);  
 }  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 printError("", arrayToSting(currentLexemeSymbols));  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_ERROR*.lexemeCode);  
 }  
  
 *// ||* else if (sourceText[textPointer] == '|') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '|') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_OR*.lexemeCode);  
 }  
  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 printError("", arrayToSting(currentLexemeSymbols));  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_ERROR*.lexemeCode);  
 }  
  
 *// = ==* else if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_EQUAL*.lexemeCode);  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_EQ*.lexemeCode);  
 }  
  
 *// ! !=* else if (sourceText[textPointer] == '!') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_NOT\_EQUAL*.lexemeCode);  
 }  
  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_NOT*.lexemeCode);  
 }  
  
 *// > >=* else if (sourceText[textPointer] == '>') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_MORE\_AND\_EQUAL*.lexemeCode);  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_MORE*.lexemeCode);  
 }  
  
 *// < <=* else if (sourceText[textPointer] == '<') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 if (sourceText[textPointer] == '=') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_LESS\_AND\_EQUAL*.lexemeCode);  
 }  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_LESS*.lexemeCode);  
 }  
  
  
 *// Символьная константа* else if (sourceText[textPointer] == '\'') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
  
 *// Если сразу вторая кавычка* if (sourceText[textPointer] == '\'') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_CONST\_CHAR*.lexemeCode);  
 }  
  
 *// Если внутри кавычки символ и после него кавычка* else if (sourceText[textPointer + 1] == '\'') {  
 currentLexemeSymbols[i++] = sourceText[textPointer++];  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_CONST\_CHAR*.lexemeCode);  
 }  
 *// Иначе - ошибка* else {  
 currentLexemeSymbols[0] = '\'';  
 printError("", arrayToSting(currentLexemeSymbols));  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_ERROR*.lexemeCode);  
 }  
 } else if (sourceText[textPointer] == ',') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_COM*.lexemeCode);  
 } else if (sourceText[textPointer] == ';') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_SEMI*.lexemeCode);  
 } else if (sourceText[textPointer] == '(') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode);  
 } else if (sourceText[textPointer] == ')') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_RIGHT\_ROUND\_BR*.lexemeCode);  
 } else if (sourceText[textPointer] == '{') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_LEFT\_BRACE*.lexemeCode);  
 } else if (sourceText[textPointer] == '}') {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_RIGHT\_BRACE*.lexemeCode);  
 } else {  
 currentLexemeSymbols[i] = sourceText[textPointer++];  
 printError("Неверный символ ", arrayToSting(currentLexemeSymbols));  
 return new LexemeModel(arrayToSting(currentLexemeSymbols), Lexeme.*T\_ERROR*.lexemeCode);  
 }  
 }  
  
 public void printError(String description, String lexeme) {  
 if (lexeme.isEmpty()) {  
 System.*out*.println("Ошибка: " + description);  
 } else {  
 System.*out*.println("Ошибка: " + description + ". Неверный символ " + lexeme + "\nСтрока " + (lineNumber + 1));  
 }  
  
 System.*exit*(1);  
 }  
  
 public int getTextPointer() {  
 return textPointer;  
 }  
  
 public void setTextPointer(int textPointer) {  
 this.textPointer = textPointer;  
 }  
  
 public int getLineNumber() {  
 return lineNumber;  
 }  
  
 public void setLineNumber(int lineNumber) {  
 this.lineNumber = lineNumber;  
 }  
  
 */\*\*  
 \* Чтение исходного кода программы из текстового файла  
 \*  
 \* @param fileName название файла  
 \*/* private void readProgramFromTextFile(String fileName) {  
 try {  
 StringBuilder theString;  
  
 File file = new File(fileName);  
 java.util.Scanner scanner = new java.util.Scanner(file);  
 theString = new StringBuilder(scanner.nextLine());  
  
 while (scanner.hasNextLine()) {  
 theString.append("\n").append(scanner.nextLine());  
 }  
  
 sourceText = (theString.toString() + "\0").toCharArray();  
 } catch (IOException e) {  
 printError("Ошибка чтения файла! " + e.toString(), "");  
 }  
 }  
  
 */\*\*  
 \* Проверка на игнорируемый символ  
 \*  
 \* @param lexemeSymb лексема в виде символа  
 \* @return флаг-результат проверки на игнорируемый символ  
 \*/* private boolean isEqualIgnoredSymbol(char lexemeSymb) {  
 for (char symbol :  
 *ignoreSymbols*) {  
 if (symbol == lexemeSymb)  
 return true;  
 }  
 return false;  
 }  
  
 private String arrayToSting(char[] list) {  
 StringBuilder result = new StringBuilder();  
 for (Character ch :  
 list) {  
 if (ch != '\u0000') {  
 result.append(ch);  
 }  
 }  
 return result.toString();  
 }  
}

Analyzer.java

package com.linguisticanalysis.analyzer;  
  
import com.linguisticanalysis.enums.Lexeme;  
import com.linguisticanalysis.scanner.Scanner;  
  
public class Analyzer {  
 private final Scanner scanner;  
  
 public Analyzer(Scanner scanner) {  
 this.scanner = scanner;  
 }  
  
 */\*\*  
 \* Проверка всей программы (аксиомы)  
 \*/* public void S() {  
 int numberOfMains = 0;  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 LexemeModel lexemeType = scanner.getNextLexeme();  
 while (lexemeType.getCode() != Lexeme.*T\_END*.lexemeCode) {  
 if (lexemeType.getCode() == Lexeme.*T\_CHAR*.lexemeCode || lexemeType.getCode() == Lexeme.*T\_INT*.lexemeCode) {  
 scanner.getNextLexeme();  
 lexemeType = scanner.getNextLexeme();  
  
 if (lexemeType.getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 A();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 B();  
 }  
  
 } else if (lexemeType.getCode() == Lexeme.*T\_VOID*.lexemeCode) {  
 lexemeType = scanner.getNextLexeme();  
 if (lexemeType.getCode() == Lexeme.*T\_MAIN*.lexemeCode) {  
 numberOfMains += 1;  
  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
  
 O();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 A();  
 }  
 }  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 lexemeType = scanner.getNextLexeme();  
 }  
  
 if (numberOfMains == 0) {  
 scanner.printError("Функция main отсутствует", lexemeType.getName());  
 } else if (numberOfMains > 1) {  
 scanner.printError("Функция main не является единственной", lexemeType.getName());  
 }  
 }  
  
 *// Проверка на лексемы* private void checkLexeme(Lexeme[] lexemes, String errorText) {  
 LexemeModel lexemeType = scanner.getNextLexeme();  
 boolean errorFlag = true;  
 for (Lexeme lexeme :  
 lexemes) {  
 if (lexemeType.getCode() == lexeme.lexemeCode) {  
 errorFlag = false;  
 break;  
 }  
 }  
 if (errorFlag) {  
 scanner.printError(errorText, lexemeType.getName());  
 }  
 }  
  
 *// Проверка на лексемы* private void checkLexeme(Lexeme lexeme, String errorText) {  
 LexemeModel lexemeType = scanner.getNextLexeme();  
 if (lexemeType.getCode() != lexeme.lexemeCode) {  
 scanner.printError(errorText, lexemeType.getName());  
 }  
 }  
  
 *// Устанавливает указатель сканера* private void setPointerTo(int pointer, int line) {  
 scanner.setTextPointer(pointer);  
 scanner.setLineNumber(line);  
 }  
  
  
 */\*\*  
 \* Описание функции  
 \*/* private void A() {  
 TF();  
 ID();  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 P();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 O();  
 }  
  
 */\*\*  
 \* Тип данных  
 \*/* private void T() {  
 checkLexeme(new Lexeme[]{Lexeme.*T\_CHAR*, Lexeme.*T\_INT*}, "Ожидался тип char или int");  
 }  
  
 */\*\*  
 \* Тип функции  
 \*/* private void TF() {  
 checkLexeme(new Lexeme[]{Lexeme.*T\_CHAR*, Lexeme.*T\_INT*, Lexeme.*T\_VOID*}, "Ожидался тип функции char, int или void");  
 }  
  
 */\*\*  
 \* Описание переменных одного типа  
 \*/* private void B() {  
 T();  
 L();  
 checkLexeme(Lexeme.*T\_SEMI*, "Ожидался символ ;");  
 }  
  
 */\*\*  
 \* Список переменных при описании  
 \*/* private void L() {  
  
 int lastPointer;  
 int lastLine;  
  
 LexemeModel lexemeType;  
  
 do {  
 ID();  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 lexemeType = scanner.getNextLexeme();  
 if (lexemeType.getCode() == Lexeme.*T\_EQ*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 I();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 }  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 } while (scanner.getNextLexeme().getCode() == Lexeme.*T\_COM*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 }  
  
 */\*\*  
 \* Блок { }  
 \*/* private void O() {  
 checkLexeme(Lexeme.*T\_LEFT\_BRACE*, "Ожидался символ {");  
  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 LexemeModel lexemeType = scanner.getNextLexeme();  
 if (lexemeType.getCode() != Lexeme.*T\_RIGHT\_BRACE*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 Z();  
 checkLexeme(Lexeme.*T\_RIGHT\_BRACE*, "Ожидался символ }");  
 }  
 }  
  
 */\*\*  
 \* Операторы  
 \*/* private void Z() {  
 LexemeModel lexemeType;  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 do {  
 setPointerTo(lastPointer, lastLine);  
 lexemeType = scanner.getNextLexeme();  
 setPointerTo(lastPointer, lastLine);  
 if (lexemeType.getCode() == Lexeme.*T\_CHAR*.lexemeCode || lexemeType.getCode() == Lexeme.*T\_INT*.lexemeCode) {  
 B();  
 } else {  
 Q();  
 }  
  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 } while (scanner.getNextLexeme().getCode() != Lexeme.*T\_RIGHT\_BRACE*.lexemeCode);  
  
 setPointerTo(lastPointer, lastLine);  
 }  
  
 */\*\*  
 \* Один оператор  
 \*/* private void Q() {  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 LexemeModel lexemeType = scanner.getNextLexeme();  
  
 if (lexemeType.getCode() == Lexeme.*T\_IF*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 IF();  
 } else {  
 if (lexemeType.getCode() == Lexeme.*T\_RETURN*.lexemeCode) {  
 A1();  
 } else {  
 if (scanner.getNextLexeme().getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 R();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 F();  
 }  
 }  
 checkLexeme(Lexeme.*T\_SEMI*, "Ожидался символ ;");  
 }  
 }  
  
 */\*\*  
 \* Присваивание  
 \*/* private void F() {  
 ID();  
 E();  
 A1();  
 }  
  
 */\*\*  
 \* Блок условий  
 \*/* private void IF() {  
  
 *// if* checkLexeme(Lexeme.*T\_IF*, "Ожидалось ключевое слово if");  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 A1();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 O();  
  
 *// else if* int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 while (scanner.getNextLexeme().getCode() == Lexeme.*T\_ELSE*.lexemeCode && scanner.getNextLexeme().getCode() == Lexeme.*T\_IF*.lexemeCode) {  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 A1();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 O();  
 lastLine = scanner.getLineNumber();  
 lastPointer = scanner.getTextPointer();  
 }  
 setPointerTo(lastPointer, lastLine);  
  
  
 *// else* lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 if (scanner.getNextLexeme().getCode() == Lexeme.*T\_ELSE*.lexemeCode) {  
 O();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 }  
 }  
  
 */\*\*  
 \* Вызов функции  
 \*/* private void R() {  
 ID();  
 checkLexeme(Lexeme.*T\_LEFT\_ROUND\_BR*, "Ожидался символ (");  
 M();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 }  
  
 */\*\*  
 \* Параметры функции  
 \*/* private void P() {  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 if (scanner.getNextLexeme().getCode() != Lexeme.*T\_RIGHT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 do {  
 T();  
 ID();  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 } while (scanner.getNextLexeme().getCode() == Lexeme.*T\_COM*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 }  
 setPointerTo(lastPointer, lastLine);  
 }  
  
 */\*\*  
 \* Список аргументов функции  
 \*/* private void M() {  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 if (scanner.getNextLexeme().getCode() != Lexeme.*T\_RIGHT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 do {  
 A1();  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 } while (scanner.getNextLexeme().getCode() == Lexeme.*T\_COM*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 }  
 setPointerTo(lastPointer, lastLine);  
 }  
  
 */\*\*  
 \* Инициал  
 \*/* private void I() {  
 checkLexeme(Lexeme.*T\_EQ*, "Ожидался символ =");  
 A1();  
 }  
  
 */\*\*  
 \* Оператор присваивания  
 \*/* private void E() {  
 Lexeme[] lexemes = new Lexeme[]{Lexeme.*T\_PLUS\_EQ*, Lexeme.*T\_SUB\_EQ*, Lexeme.*T\_DIV\_EQ*, Lexeme.*T\_MUL\_EQ*, Lexeme.*T\_EQ*};  
 checkLexeme(lexemes, "Ожидался оператор присваивания");  
 }  
  
 */\*\*  
 \* Идентификатор (име переменной или функции)  
 \*/* private void ID() {  
 checkLexeme(Lexeme.*T\_ID*, "Ожидалось имя перемнной или функции");  
 }  
  
 */\*\*  
 \* Логические операции  
 \*/* private void A1() {  
 int lastPointer;  
 int lastLine;  
 do {  
 do {  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 }  
 while (scanner.getNextLexeme().getCode() == Lexeme.*T\_NOT*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 A2();  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 } while (scanner.getNextLexeme().getCode() == Lexeme.*T\_AND*.lexemeCode || scanner.getNextLexeme().getCode() == Lexeme.*T\_OR*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 }  
  
 */\*\*  
 \* Операции сравнения  
 \*/* private void A2() {  
 int lastPointer;  
 int lastLine;  
  
 A3();  
  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 LexemeModel lexemeType = scanner.getNextLexeme();  
 if (lexemeType.getCode() == Lexeme.*T\_MORE*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_LESS*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_MORE\_AND\_EQUAL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_LESS\_AND\_EQUAL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_EQUAL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_NOT\_EQUAL*.lexemeCode) {  
 A3();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 }  
 }  
  
 */\*\*  
 \* Арифметические операции  
 \*/* private void A3() {  
 int lastPointer;  
 int lastLine;  
 LexemeModel lexemeType;  
 do {  
 A4();  
 lastPointer = scanner.getTextPointer();  
 lastLine = scanner.getLineNumber();  
 lexemeType = scanner.getNextLexeme();  
 } while (lexemeType.getCode() == Lexeme.*T\_SUB*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_MUL*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_PLUS*.lexemeCode ||  
 lexemeType.getCode() == Lexeme.*T\_DIV*.lexemeCode);  
 setPointerTo(lastPointer, lastLine);  
 }  
  
 */\*\*  
 \* Значение (переменная, константы, результат вызова функции, выражение)  
 \*/* private void A4() {  
 int lastPointer = scanner.getTextPointer();  
 int lastLine = scanner.getLineNumber();  
 LexemeModel lexemeType = scanner.getNextLexeme();  
  
 if (lexemeType.getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 A1();  
 checkLexeme(Lexeme.*T\_RIGHT\_ROUND\_BR*, "Ожидался символ )");  
 } else if (lexemeType.getCode() == Lexeme.*T\_ID*.lexemeCode) {  
 if (scanner.getNextLexeme().getCode() == Lexeme.*T\_LEFT\_ROUND\_BR*.lexemeCode) {  
 setPointerTo(lastPointer, lastLine);  
 R();  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 ID();  
 }  
 } else {  
 setPointerTo(lastPointer, lastLine);  
 Lexeme[] lexemes = new Lexeme[]{Lexeme.*T\_CONST\_CHAR*, Lexeme.*T\_CONST\_INT*};  
 checkLexeme(lexemes, "Ожидалась целочисленная или символьная константа");  
 }  
 }  
}

Main.java

import com.linguisticanalysis.analyzer.Analyzer;  
import com.linguisticanalysis.scanner.Scanner;  
  
public class Main {  
  
 public static void main(String[] args) {  
 Scanner scan = new Scanner("/Users/egor/Desktop/untitled.txt");  
 Analyzer analyzer = new Analyzer(scan);  
 analyzer.S();  
 System.*out*.println("Ошибок не обнаружено!");  
 }  
}