**Министерство образования и науки Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего профессионального образования**

**«Московский государственный технический университет имени Н.Э. Баумана»**

**(МГТУ им. Н.Э.Баумана)**

ФАКУЛЬТЕТ \_\_\_\_\_\_\_\_\_\_\_Информатика и системы управления \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_Программное обеспечение ЭВМ и информационные технологии\_\_\_

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОЙ РАБОТЕ***

***НА ТЕМУ:***

***Построение синтаксического дерева программ на языке Cool***

Студент \_\_ИУ7-21М\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ \_Яковлева О.В.\_\_\_

(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_ \_Ступников А.А.\_

(Подпись, дата) (И.О.Фамилия)

*2020 г.*

**Оглавление**

[Введение 3](#_Toc51152897)

[1. Аналитический раздел 4](#_Toc51152898)

[1.1. Основы принципов построения компиляторов 4](#_Toc51152899)

[1.2. Особенности языка программирования Cool 6](#_Toc51152900)

[2. Конструкторский раздел 7](#_Toc51152901)

[2.1. Грамматика языка программирования Cool 7](#_Toc51152902)

[2.2. Построение абстрактного синтаксического дерева 7](#_Toc51152903)

[3. Технологический раздел 9](#_Toc51152904)

[3.1. Выбор языка программирования и средств разработки 9](#_Toc51152905)

[3.2. Описание классов – узлов АСТ 10](#_Toc51152906)

[3.3. Преобразование КСТ в АСТ 11](#_Toc51152907)

[3.4. Примеры работы программы построения АСТ 13](#_Toc51152908)

[Заключение 15](#_Toc51152909)

[Список используемых источников 16](#_Toc51152910)

[Приложение 1 17](#_Toc51152911)

**Введение**

Целью курсового проекта является разработка программы построения синтаксического дерева программ, написанных на языке программирования Cool.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Изучить основы принципов построения компиляторов.
2. Реализовать лексический и синтаксический анализаторы.
3. Подготовить набор тестовых данных для тестирования и демонстрации полученного решения.
4. **Аналитический раздел**
   1. **Основы принципов построения компиляторов**

Компилятор – это программа, которая считывает текст программы на одном языке, называемом исходным, и транслирует или переводит его в эквивалентный текст на другом языке – целевом (рисунок 1.1). Одна из важных ролей компилтора состоит в сообщении об ошибках в исходной программе, обнаруженных в процессе трансляции.



Рисунок 1.1. Компилятор

Отображение компилятором исходной программы в семантически эквивалентную ей целевую программу разделяется на две части: анализ и синтез [1].

Анализ разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Если анализ обнаруживает, что исходная программа неверно составлена синтаксически либо дефектна семантически, он должен выдавать информативные сообщения об этом, чтобы пользователь мог исправить обнаруженные ошибки.

Синтез строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов.

Как правило, анализ называют начальной стадией (front-end), а синтез – заключительной (back-end).

Фазы компиляции представлены на рисунке 1.2.

|  |
| --- |
|  |
| Рисунок 1.2. Фазы компиляции |

В рамках данной курсовой работы необходимо реализовать первые две фазы из вышеперечисленных.

Лексический анализ является первой фазой компиляции. Также данная фаза называется сканированием. Лексический анализатор читает поток символов, которые составляют исходную программу, и группирует их в значащие последовательности, называемые лексемами. Для каждой построенной лексемы анализатор строит выходной токен вида [1]:

(имя\_токена, значение\_атрибута)

Первый компонент токена представляет собой абстрактный символ, который используется во время синтаксического анализа, а второй компонент указывает на запись в таблице символов, которая соответствует данному токену.

Вторая фаза компиляции – синтаксический анализ или разбор. Анализатор использует первые компоненты токенов, полученных при лексическом анализе, для построения промежуточного древовидного представления, которое описывает грамматическую структуру потока токенов. Обычно для этой цели используется синтаксическое дерево, в котором каждый внутренний узел представляет операцию, а дочерние узлы – аргументы этой операции.

* 1. **Особенности языка программирования Cool**

Cool — Classroom Object-Oriented Language является учебным языком программирования, включающий классы и объекты. Cool является строготипизированным языком, т.е. проверка типов происходит статически на этапе компиляции. Программа по сути является набором классов. Классы включают в себя набор полей и функций (feature). Все поля являются видимыми внутри базового и производного классов. Все функции являются видимыми отовсюду (public). Данный язык построен на выражениях (expression). Все функции в качестве аргументов могут принимать выражения и телами всех функций являются выражения, который возвращают какой-либо результат. Даже такие функции, как вывод строки, возвращают результат, а именно экземпляр класса, из которого они вызваны. Cool содержит также и статические классы String, Int, Bool, от которых нельзя наследоваться и которые не могут принимать значения null. Объекты данных классов передаются по значению, а не по ссылке [2].

1. **Конструкторский раздел**
   1. **Грамматика языка программирования Cool**

Грамматика языка Cool содержит следующие синтаксические конструкции [3]:

* Программа, состоящая из набора классов, разделяемых точкой с запятой;
* Класс;
* Поле и метод класса;
* Аргумент функции;
* Выражения:
  + Вызов метода другого класса;
  + Вызов метода текущего класса;
  + Условное выражение;
  + Конструктор класса;
  + Арифметические операции (умножение, деление, сложение, вычитание);
  + Операции сравнения;
  + Константы (числовые, строковые, булевы)
  + Операции присваивания;
  + Блок выражений, заключенный в фигурные скобки.

Грамматика языка Cool приведена в Приложении 1 в форме РБНФ.

* 1. **Построение абстрактного синтаксического дерева**

Построение абстрактного синтаксического дерева будет происходить в четыре этапа.

На первом этапе с помощью генератора создаются лексический и синтаксический анализаторы. На втором и третьем этапах происходят лексический и синтаксический анализы, в результате чего создается конкретное синтаксическое дерево (КСТ) – поток токенов, представленный в виде дерева, в котором содержится вся информация (или большая ее часть) о программе ввода, обычно совпадающая с тем, что описано в грамматике языка. На четвертом этапе создается абстрактное синтаксическое дерево (АСТ). Для этого требуется на основе грамматики создать набор классов – узлов АСТ, каждый из которых будет описывать один оператор языка Cool. Далее требуется обойти полученное на третьем этапе КСТ и редуцировать его, преобразовав его узлы в узлы АСТ.

На рисунке 2.1 приведена диаграмма построения АСТ в нотации IDEF0.

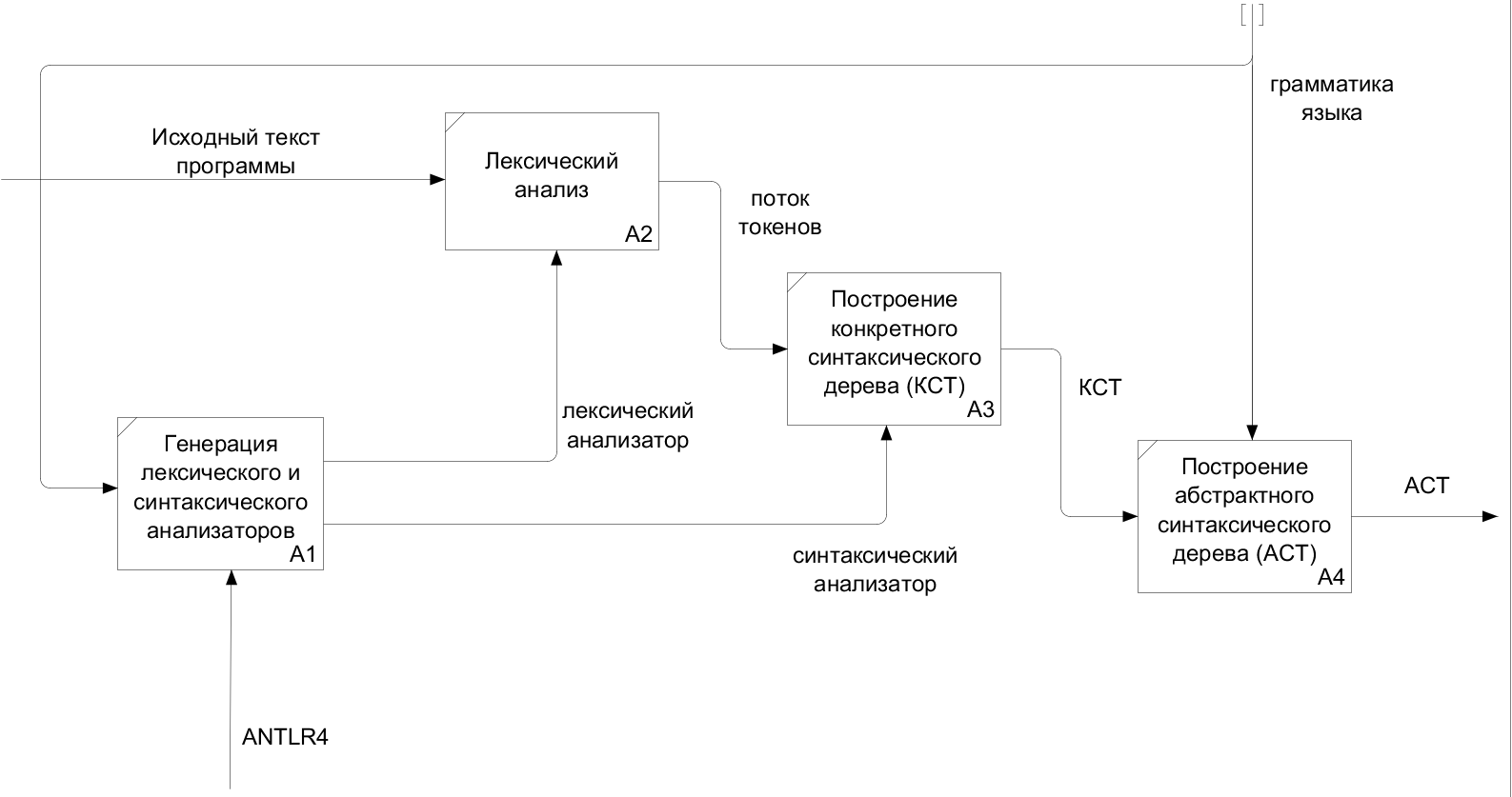


Рисунок 2.1. Построение абстрактного синтаксического дерева.

1. **Технологический раздел**
   1. **Выбор языка программирования и средств разработки**

В качестве языка программирования для разработки был выбран объектно-ориентированный язык Java. В качестве среды разработки выбрана среда IntelliJ IDEA, так как она позволяет быстро и удобно вести разработку за счет автоматического дополнения кода, удобного рефакторинга и поддержки новейших технологий и фреймворков.

Для лексического анализа и построения конкретного синтаксического дерева выбран инструмент ANTLR.

ANTLR – мощный генератор лексического и синтаксического анализаторов для чтения, обработки, выполнения или перевода структурированных текстовых или двоичных файлов.

Данный генератор является [LL(\*)](https://ru.wikipedia.org/wiki/LL-%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80), он существует уже более 20 лет, а в 2013 год вышла его 4-я версия. Сейчас его разработка ведется на GitHub. В данный момент он позволяет генерировать синтаксические анализаторы на языках Java, C#, Python2, Python3, JavaScript. Также в этом инструменте достаточно просто разрабатывать и отлаживать грамматики. Несмотря на то, что LL грамматики не допускают леворекурсивных правил, в ANTLR, начиная с 4-й версии, появилась возможность записи таких правил (за исключением правил со скрытой или косвенной левой рекурсией). При генерации синтаксического анализатора происходит преобразование таких правил в обычные не леворекурсивные. [4].

Кроме того, в 4-й версии значительно улучшена производительность синтаксического анализа благодаря использованию алгоритма Adaptive LL(\*). Данный алгоритм совмещает преимущества относительно медленных и непредсказуемых алгоритмов GLL и [GLR](https://en.wikipedia.org/wiki/GLR_parser), которые, однако, способны разрешать случаи с неоднозначностью (используются при разборе естественных языков), и стандартных, быстрых LL-алгоритмов рекурсивного спуска, которые, в свою очередь, не способны разрешать все задачи с неоднозначностью. Суть алгоритма заключается в псевдопараллельном запуске LL-синтаксических анализаторов на каждом правиле, их кэшировании и выборе подходящего предсказания (в отличие от GLR, где допустимо несколько альтернатив). Таким образом, алгоритм является динамическим. Несмотря на то, что наихудшая теоретическая сложность алгоритма является O(n4), по факту скорость синтаксического анализа для существующих языков программирования является линейной. Также в четвертой версии сильно эволюционировала возможность восстановления процесса синтаксического анализа после обнаружения синтаксических ошибок [4].

* 1. **Описание классов – узлов АСТ**

В листинге 1 приведен пример классов, описывающих узлы АСТ “программа”, “класс”, “поле класса” и “метод класса”. Полем класса “программа” является список элементов типа “класс”, так как программы на языке Cool представляют собой список классов. Полями класса “класс” являются его название, название родительского класса и список полей и методов. Классы, описывающие узлы “поле класса” и “метод класса” наследуются от одного базового класса. Полями класса “метод класса” являются его название, список аргументов, тип возвращаемого значения и тело. Полями класса “поле класса” его название, тип и значение.

Таким образом, каждая программа на языке Cool имеет иерархическую структуру. Эта структура описывается на основании грамматики в виде классов Java с использованием свойств ООП – наследования, полиморфизма и инкапсуляции. При этом синтаксис самого Cool не имеет значения – узлы абстрактного синтаксического дерева не содержат ключевых слов языка, разделителей операторов, скобок и прочего.

Листинг 1. Некоторые узлы АСТ.

public static class Program extends ASTNode {  
 public List<Class> classes;  
 public Program(List<Class> c) {  
 classes = c;  
 }  
 public Program() {}  
}

public static class Class extends ASTNode {  
 public String name;  
 public String parent;  
 public List<Feature> features;  
 public Class(String n, String p, List<Feature> fs) {  
 name = n;  
 parent = p;  
 features = fs;  
 }  
 public Class() {}  
}

public static class Feature extends ASTNode {  
 public Feature() {  
 }  
}

public static class Method extends Feature {  
 public String name;  
 public List<Argument> arguments;  
 public String typeid;  
 public Expression body;  
 public Method(String n, List<Argument> f, String t, Expression b) {  
 name = n;  
 arguments = f;  
 typeid = t;  
 body = b;  
 }  
 public Method() {}  
}  
  
public static class Property extends Feature {  
 public String name;  
 public String typeid;  
 public Expression value;  
 public Property(String n, String t, Expression v) {  
 name = n;  
 typeid = t;  
 value = v;  
 }  
 public Property() {}  
}

* 1. **Преобразование КСТ в АСТ**

Для обхода КСТ ANTLR4 предоставляет два типовых подхода (паттерна проектирования): Visitor и Listener. Оба подхода позволяют вынести логику обработки каждого типа узла в отдельные методы.

В Visitor для обработки потомков какого-либо узла необходимо вручную вызывать методы их обхода. При этом если родитель имеет три потомка, и вызвать методы только для двух узлов, то часть поддерева вообще не будет обработана. В Listener (Walker) же методы посещения всех потомков вызываются автоматически. В Listener существует метод, вызываемый в начале посещения узла (enterNode) и метод, вызываемый после посещения узла (exitNode). Эти методы также можно реализовать с помощью механизма событий. Методы Visitor, в отличие от Listener, могут возвращать объекты и даже могут быть типизированными, т.е. каждый метод visit, будет возвращать объект ASTNode, который является общим предком для всех остальных узлов АСТ.

В данной курсовой работе для обхода дерева будет использоваться Visitor. Так код преобразования дерева получается более функциональным и лаконичным за счет того, что в нем нет необходимости хранить информацию о посещенных узлах.

В листинге 2 представлен пример метода visit посещения класса Add, описывающего оператор добавления. Извлекаются потомки узла КСТ и для них вызывается метод visit. Затем создается узел АСТ – операция сложения, в конструктор класса которого передаются два операнда.

Листинг 2. Метод посещения узла КСТ на примере оператора Add

**public class** CoolBaseVisitorImpl<T> **extends** CoolBaseVisitor<CoolAST.ASTNode> {

@Override  
**public** CoolAST.ASTNode visitAdd(CoolParser.AddContext ctx) {  
 CoolAST.ASTNode left = visit(ctx.getChild(0));  
 CoolAST.ASTNode right = visit(ctx.getChild(2));  
 **return new** CoolAST.Add((CoolAST.Expression) left, (CoolAST.Expression) right);  
}

…

}

Таким образом, начиная построение дерева с посещения корневого узла “программа”, последовательно вызываем методы visit для посещения потомков узла. Каждый метод visit преобразует узел КСТ в АСТ.

* 1. **Примеры работы программы построения АСТ**

Примеры работы программы построения АСТ приведены в таблице 1.

Таблица 1. Примеры работы программы построения АСТ

|  |  |
| --- | --- |
| Исходный код | Синтаксическое дерево |
| class Math  {   lower(n: Int, m: Int) : Int {  if n < m  then n  else m  };  }; | {  "nodetype" : "Program",  "classes" : [ {  "nodetype" : "Class",  "name" : "Math",  "parent" : null,  "features" : [ {  "nodetype" : "Method",  "name" : "lower",  "arguments" : [ {  "nodetype" : "Argument",  "name" : "n",  "typeid" : "Int"  }, {  "nodetype" : "Argument",  "name" : "m",  "typeid" : "Int"  } ],  "typeid" : "Int",  "body" : {  "nodetype" : "If",  "predicate" : {  "nodetype" : "LessThan",  "e1" : {  "nodetype" : "Terminal",  "name" : "n"  },  "e2" : {  "nodetype" : "Terminal",  "name" : "m"  }  },  "ifbody" : {  "nodetype" : "Terminal",  "name" : "n"  },  "elsebody" : {  "nodetype" : "Terminal",  "name" : "m"  }  }  } ]  } ]  } |

|  |  |
| --- | --- |
| Исходный код | Синтаксическое дерево |
| class Math  {   add(n: Int, m: Int) : Int {  n + m;  };  }; | {  "nodetype" : "Program",  "classes" : [ {  "nodetype" : "Class",  "name" : "Math",  "parent" : null,  "features" : [ {  "nodetype" : "Method",  "name" : "add",  "arguments" : [ {  "nodetype" : "Argument",  "name" : "n",  "typeid" : "Int"  }, {  "nodetype" : "Argument",  "name" : "m",  "typeid" : "Int"  } ],  "typeid" : "Int",  "body" : {  "nodetype" : "Add",  "e1" : {  "nodetype" : "Terminal",  "name" : "n"  },  "e2" : {  "nodetype" : "Terminal",  "name" : "m"  }  }  } ]  } ]  } |

Продолжение таблицы 1

**Заключение**

В ходе выполнения данной курсовой работы были изучены основы принципов построения компиляторов и реализована программа, преобразующая код программ на языке Cool в абстрактное синтаксическое дерево. Программа предназначена для выполнения первых двух фаз компиляции – лексического и синтаксического анализов. Лексический и синтаксический анализаторы сгенерированы с помощью инструмента ANTLR4. Реализовано преобразование конкретного синтаксического дерева, которое строится сгенерированным синтаксическим анализатором, в абстрактное. Подготовлены примеры программ на языке Cool для тестирования и демонстрации полученного решения.

**Список используемых источников**

1. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж. Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008;
2. Cool [Электронный ресурс]. URL: http://progopedia.ru/language/cool/ (дата обращения: 16.09.2020)
3. Грамматика языка Cool [Электронный ресурс]. URL: https://github.com/antlr/grammars-v4/tree/master/cool (дата обращения: 16.09.2020)
4. Теория и практика парсинга исходников с помощью ANTLR и Roslyn. [Электронный ресурс]. URL: https://habr.com/ru/company/pt/blog/210772/ (дата обращения: 16.09.2020)

**Приложение 1**

**Грамматика языка Cool в формате РБНФ (ANTLR)**

**grammar** Cool;  
  
program  
 : programBlocks  
 ;  
  
programBlocks  
 : classDefine **';'** programBlocks # classes  
 | EOF # eof  
 ;  
  
classDefine  
 : CLASS TYPEID (INHERITS TYPEID)? **'{'** (feature **';'**)\* **'}'** ;  
  
feature  
 : OBJECTID **'('** (formal (**','** formal)\*)? **')' ':'** TYPEID **'{'** expression **'}'** # method  
 | OBJECTID **':'** TYPEID (ASSIGNMENT expression)? # property  
 ;

*/\* method argument \*/*  
formal  
 : OBJECTID **':'** TYPEID  
 ;  
expression  
 : expression (**'@'** TYPEID)? **'.'** OBJECTID **'('** (expression (**','** expression)\*)? **')'** # methodCall  
 | OBJECTID **'('** (expression (**','** expression)\*)? **')'** # ownMethodCall  
 | IF expression THEN expression ELSE expression FI # if  
 | WHILE expression LOOP expression POOL # while  
 | **'{'** (expression **';'**)+ **'}'** # block  
 | LET OBJECTID **':'** TYPEID (ASSIGNMENT expression)? (**','** OBJECTID **':'** TYPEID (ASSIGNMENT expression)?)\* IN expression # letIn  
 | CASE expression OF (OBJECTID **':'** TYPEID CASE\_ARROW expression **';'**)+ ESAC # case  
 | NEW TYPEID # new  
 | INTEGER\_NEGATIVE expression # negative  
 | ISVOID expression # isvoid  
 | expression MULTIPLY expression # multiply  
 | expression DIVISION expression # division  
 | expression ADD expression # add  
 | expression MINUS expression # minus  
 | expression LESS\_THAN expression # lessThan  
 | expression LESS\_EQUAL expression # lessEqual  
 | expression EQUAL expression # equal  
 | NOT expression # boolNot  
 | **'('** expression **')'** # parentheses  
 | OBJECTID # id  
 | INT # int  
 | STRING # string  
 | TRUE # true  
 | FALSE # false  
 | OBJECTID ASSIGNMENT expression # assignment  
 ;  
 *// key words*CLASS  
 : C L A S S  
 ;  
  
ELSE  
 : E L S E  
 ;  
  
FALSE  
 : **'f'** A L S E  
 ;  
  
FI  
 : F I  
 ;  
  
IF  
 : I F  
 ;  
  
IN  
 : I N  
 ;  
  
INHERITS  
 : I N H E R I T S  
 ;  
  
ISVOID  
 : I S V O I D  
 ;  
  
LET  
 : L E T  
 ;  
  
LOOP  
 : L O O P  
 ;  
  
POOL  
 : P O O L  
 ;  
  
THEN  
 : T H E N  
 ;  
  
WHILE  
 : W H I L E  
 ;  
  
CASE  
 : C A S E  
 ;  
  
ESAC  
 : E S A C  
 ;  
  
NEW  
 : N E W  
 ;  
  
OF  
 : O F  
 ;  
  
NOT  
 : N O T  
 ;  
  
TRUE  
 : **'t'** R U E  
 ;  
 *// primitives*STRING  
 : **'"'** (ESC | ~ ["\\])\* **'"'** ;  
INT  
 : [0-9]+  
 ;  
TYPEID  
 : [A-Z] [\_0-9A-Za-z]\*  
 ;  
OBJECTID  
 : [a-z] [\_0-9A-Za-z]\*  
 ;  
ASSIGNMENT  
 : **'<-'** ;  
CASE\_ARROW  
 : **'=>'** ;  
ADD  
 : **'+'** ;  
MINUS  
 : **'-'** ;  
MULTIPLY  
 : **'\*'** ;  
DIVISION  
 : **'/'** ;  
LESS\_THAN  
 : **'<'** ;  
LESS\_EQUAL  
 : **'<='** ;  
EQUAL  
 : **'='** ;  
INTEGER\_NEGATIVE  
 : **'~'** ;  
**fragment** A  
 : [aA]  
 ;  
**fragment** C  
 : [cC]  
 ;  
**fragment** D  
 : [dD]  
 ;  
**fragment** E  
 : [eE]  
 ;  
**fragment** F  
 : [fF]  
 ;  
**fragment** H  
 : [hH]  
 ;  
**fragment** I  
 : [iI]  
 ;  
**fragment** L  
 : [lL]  
 ;  
**fragment** N  
 : [nN]  
 ;  
**fragment** O  
 : [oO]  
 ;  
**fragment** P  
 : [pP]  
 ;  
**fragment** R  
 : [rR]  
 ;  
**fragment** S  
 : [sS]  
 ;  
**fragment** T  
 : [tT]  
 ;  
**fragment** U  
 : [uU]  
 ;  
**fragment** V  
 : [vV]  
 ;  
**fragment** W  
 : [wW]  
 ;  
**fragment** ESC  
 : **'\\'** (["\\/bfnrt] | UNICODE)  
 ;  
**fragment** UNICODE  
 : **'u'** HEX HEX HEX HEX  
 ;  
**fragment** HEX  
 : [0-9a-fA-F]  
 ;

*// comments*OPEN\_COMMENT  
 : **'(\*'** ;  
CLOSE\_COMMENT  
 : **'\*)'** ;  
COMMENT  
 : OPEN\_COMMENT (COMMENT | .)\*? CLOSE\_COMMENT -> skip  
 ;  
ONE\_LINE\_COMMENT  
 : **'--'** (~ **'\n'**)\* **'\n'**? -> skip  
 ;

*// skip spaces, tabs, newlines, note that \v is not suppoted in antlr*WHITESPACE  
 : [ \t\r\n\f]+ -> skip  
 ;