**Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

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

**(национальный исследовательский университет)» (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:***

***Разработка генератора синтаксических анализаторов и генерация промежуточного кода для контекстно-свободной грамматики.***

Студент ИУ7-21М

**Осипов О. К.**

(Подпись, дата)

Руководитель курсовой работы **Ступников А. А.**

(Подпись, дата)

Консультант

(Подпись, дата)

*2020 г.*

**Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

**З А Д А Н И Е**

УТВЕРЖДАЮ

Заведующий кафедрой ИУ-7

Рудаков И.В.

« » 20 г.

**на выполнение курсовой работы**

по дисциплине *конструирование компиляторов*

Студент группы *ИУ7-21М Осипов Олег Константинович*

Тема курсовой работы *Разработка генератора синтаксических анализаторов и генерация промежуточного кода для контекстно-свободной грамматики.*

Направленность КР *учебная*

Источник тематики *НИР*

График выполнения работы: 25% к 3 нед., 50% к 9 нед., 75% к 12 нед., 100% к 15 нед.

**Задание** провести анализ предметной области: изучить существующие программные средства, создающие синтаксические анализаторы, изучить специфику входного формата грамматики. Ознакомиться с основными методами и алгоритмами синтаксического разбора, а также построение программ анализаторов. Изучить распространённые представления промежуточного кода, выбрать лучшее из них. Разработать программу, генерирующую синтаксические анализаторы, предназначенные для выполнения синтаксического анализа в ходе компиляции программы. Синтаксический анализатор должен иметь модуль лексического анализа, для получения токена, т.е. грамматического терминального символа контекстно-свободной грамматики. Описание грамматики языка должно быть оформлено в виде текстового файла в формате JSON. Необходимо также определить структуру JSON-документа для представления грамматики. Определить формат таблиц символов, содержащих информацию о идентификаторах, константах, найденных во входном потоке. Разработать процедуру генерации промежуточного кода, с учётом семантических проверок (проверка типов, проверка использования переменных после их объявлений) для заданной контекстно-свободной грамматики.

***Оформление курсовой работы:***

Расчетно-пояснительная записка на 50 листах формата А4. Дата выдачи задания « » 20 20г.

**Руководитель курсовой работы** Ступников А.А.

(Подпись, дата)

**Студент** Осипов О.К.

(Подпись, дата)

**Содержание.**

**1. Аналитический раздел………………………………………………………….4.**

**1.1 Цели и задачи работы……………………………………………………4.**

**1.2. Основные понятия о грамматиках…………………………………….4.**

**1.3. Существующие аналоги………………………………………………...6.**

**1.4. Входной формат грамматики…………………………………………..11.**

**1.5 Формат промежуточного кода………………………………………….18.**

**1.6. Исходная грамматика входного языка.................................................20.**

**2. Конструкторский раздел.**

**2.1. Основная структура генератора…………………………………….30.**

**2.2 Модуль разбора JSON-документов………………………………… 31.**

**2.3. Модуль обработки JSON-дерева…………………………………….. 33.**

**2.4. Модуль генератора лексического анализатора…………………… 37.**

**2.5 Организация таблицы символов…………………………………….. 43.**

**2.6. Модуль синтаксического анализа…………………………………. 49.**

**2.7. Модуль семантического анализа и генерации кода……………….. 53.**

**3. Технологический раздел.**

**3.1. Сборка и запуск приложения…………………………………………. 26.**

**Список литературы…………………………………………………………………27.**

**Приложения………………………………………………………………………… 28.**

**1. Аналитический раздел.**

**1.1. Цели и задачи данной работы.**

*Цель работы:* Изучение теоретических основ конструирования компиляторов, основных понятий и теории синтаксического анализа, а также ознакомление с назначением и принципам работы лексического анализатора. Изучение основных методов семантического анализа, понятий синтаксически-управляемого перевода и схем синтаксически управляемой трансляции, а также методов генерации промежуточного кода. Разработка программного обеспечения, способное по заданному входному формату грамматики создать лексический и синтаксический анализаторы. Кроме того, данная программа должна генерировать промежуточный, машинно-независимый код, а перед генерацией кода осуществить семантические проверки.

**1.2. Основные понятия о грамматиках.**

Формальные языки могут быть определены грамматиками. Существуют два типа грамматик: порождающие и распознающие [1]. Любая порождающая грамматика характеризуется множеством терминалов, нетерминалов, правилами вывода, а также первым или начальным правилом вывода. Ниже дано формальное определение грамматики.

**Грамматикой *G*** называется четвёрка где:

* *N –* конечное множество *нетерминальных символов*, или

*нетерминалов (*иногда называемых вспомогательными символами, синтаксическими переменными);

* *T –* не пересекающееся с *N* конечное множество *терминальных*

*символов* или *терминалов*;

* *P –* конечное подмножество множества:

(элемент множества *P* называется *правилом* (или *продукцией*) и обозначается как ;

* *S –* выделенный символ из *N,* называемый *стартовым*

(*начальным*) *символом* грамматики*.*

Левую часть правила (первый элемент из пары ) называют *заголовком* правила (продукции), а правую часть – телом.

Для одной и той же левой части правила может существовать множество строк . Такие строки называются *альтернативами.* При наличии альтернатив можно использовать ниже приведённую сокращённую запись.

Пример 1.1.

Для множества правил

правила можно представить в следующем виде:

Грамматика порождает множество *выводимых цепочек,* состоящих из грамматических символов – терминалов и нетерминалов. Процесс порождения состоит в получении такого множества. Он определён следующим рекурсивным способом:

1) *S* – выводимая цепочка.

2) Если – выводимая цепочка и существует правило то

– тоже является выводимой цепочкой.

Выводимая цепочка грамматики *G,* состоящая только из терминальных символов называется *терминальной цепочкой* или *предложением.*

**Язык *L(G)*** порождаемый грамматикой *G –* это множество предложений данной грамматики.

Согласно иерархии Хомского, существуют следующие типы грамматик:

(1) **Праволинейной грамматикой *G*** называется грамматика, в которой каждое её правило из множества *P* имеет вид: Иными словами, левые части всех правил состоят ровно из одного нетерминала, а правые части представляют из себя либо строку из терминалов, либо строку из терминалов и нетерминала, стоящего в самом конце правила.

(2) **Контекстно-свободной грамматикой *G*** называют грамматику, в которой каждое её правило из множества *P* имеет вид: .

Иными словами, левые части всех правил такой грамматики состоят ровно из одного нетерминала.

(3) **Контекстно-зависимой грамматикой *G*** называют грамматику, в которой каждое её правило из множества *P* имеет вид: , и не существует правил вида .

(4) **Грамматикой общего вида *G***(*грамматика без ограничений*)называются грамматики, не удовлетворяющие ни одному из условий (1), (2), (3). Отметим, что контекстно-свободная грамматика, содержащая правила вида , не является контекстно-зависимой. Грамматики типа (1) являются также грамматиками типа (2), (3) (если нет -правил). Т.е. каждая грамматика лежащая выше других, подходит под типы, определённые ниже, но не наоборот (не все контекстно-свободные языки являются регулярными).

Здесь и далее под грамматикой будет пониматься именно контекстно-свободная грамматика. Имена терминальных символов грамматики будут обозначаться жирным шрифтом. Имена нетерминалов будут обозначаться в верхнем регистре. Имя стартового символа грамматики будет обозначаться как *S*, если не указано иное.

**1.3 Существующие аналоги.**

На сегодняшний день существуют программные средства, позволяющие по произвольно заданной контекстно-свободной грамматике создать программу синтаксического анализатора. Наиболее известные из них это ANTLR (Another Tool for Language Recognition), и GNU Yacc.

**ANTLR.**

Библиотека ANTLR имеет широкие возможности по созданию анализатора, в частности, она позволяет создавать лексические анализаторы, а также генерировать программу анализатора на определённый язык программирования: C++, Java, C#, Python и т.д. Данное средство использует нисходящий анализ при построении анализатора. К сожалению, постройка хотя бы другого класса анализаторов, например, восходящих, данная программа не умеет. Кроме того, данная программа не способна разрешать неявную левую рекурсию в грамматиках. Решение данной задачи полностью возлагается на разработчика языка.

Файл грамматики имеет расширение .g4 и состоит из следующих разделов:

* Заголовок имени грамматики. Состоит из ключевого слова *grammar* и следующим за ним идентификатора. Заканчивается символом точка с запятой, следующим за идентификатором. Имя грамматики и имя файла должны совпадать;
* Раздел опций (options {…}). Данный раздел не является обязательным;
* Раздел токенов (tokens {…}). В данном разделе перечисляются имена токенов через запятую. Токены являются терминальными символами грамматики, для которых не определены лексические правила. Имена токенов прописные. Данный раздел не является обязательным;
* Раздел инструкций импорта. Не является обязательным;
* Раздел синтаксически управляемых действий (actions {…}). Содержит исполняемый Java-код. Не является обязательным;
* Раздел правил (синтаксических и лексических). Имя для лексического правила набирается в прописном регистре. Имя синтаксического правила должно начинаться со строчной буквы.

Правила грамматики могут быть представлены в следующих видах:

*LEXRULE : lexems ;*

*syntaxRule : elements ;*

Здесь, *LEXRULE* – имя лексического правила, задающего шаблон для терминала (с именем *LEXRULE*), *lexems* – элементы лексического правила, разделённые пробелами между собой. Элементами лексического правила могут быть последовательности символов, заключённых в одинарных кавычках, имена других лексических правил, или *фрагменты*. Фрагмент – это часть грамматики, которая может быть только частью лексического правила. Фрагменты нельзя использовать в синтаксических правилах. Лексическое правило может стать фрагментом. Для этого необходимо предварить имя правила ключевым словом fragment. Ключевое слово указывается перед именем правила отдельно.

Имя *syntaxRule* обозначает продукцию в грамматике с одноимённым нетерминалом в качестве заголовка и *elements* в качестве тела. Здесь *elements* обозначает последовательность имён правил (любых, как лексических так и синтаксических) и строк символов, заключенных в одинарные кавычки.

Каждое правило заканчивается символом «точка с запятой» и отделяется от другого правила символом новой строки. Первое определённое правило является продукцией для стартового символа грамматики. ANTLR также позволяет группировать грамматические элементы в цепочки. ANTLR предоставляет собственные операторы повторения, перечисленные ниже:

1. “+” – оператор повторения 1 или более раз;
2. “\*” – оператор повторения 0 или более раз;
3. “?” – оператор повторения 0 или 1 раз.

Каждый из выше перечисленных операторов повторяет цепочку грамматических символов, расположенную слева от него. Ниже описана грамматика арифметических выражений в форме синтаксиса ANTLR.

**grammar** Expressions;

prog: (expr NEWLINE)+ ;

expr: expr ( '\*' | '/' ) term | term ;

term: term ('+' | '-' ) factor | factor ;

factor: NUMBER | '(' expr ')' ;

NEWLINE : [\n];

NUMBER : [0-9]+(\.[0-9]+)? ;

Листинг 1. Грамматика арифметических выражении в нотации ANTLR.

Формат описания грамматики, предлагаемый ANTLR, обладает большим недостатком. В первую очередь, от регистра имени зависит тип правила (определение терминала, или синтаксической продукции) для грамматики. Зависимость от регистра может внести путаницу при создании грамматики. Лучшим решением является определение отдельной секции для лексических правил.

**GNU Yacc.**

Ещё одним средством для создания анализатора является программное средство GUN *yacc* в совокупности с *lex*. И *yacc*, и *lex* имеют формат описания, отличный от формата ANTLR. Причём *lex* - это средство для создания лексических анализаторов, и оно используется вместе с *yacc*, поскольку синтаксический анализатор читает токены от лексического анализатора, а не входные строки исходного текстового файла.

Входной файл грамматики для приложения GNU Bison имеет имя, оканчивающееся на *.y,* Он состоит из четырёх секций, расположенных в порядке, указанном ниже:

* Секция пролога. Содержит макроопределения, макросы, объявления переменных и функций на языке C. Может отсутствовать;
* Секция определений. В данной секции определяются имена терминальных и нетерминальных символов грамматики. Для объявления терминала необходимо объявить токен с его именем. Имя токена должно быть полностью в верхнем регистре. Токены объявляются инструкцией *%token*. Секция не является обязательной;
* Секция правил. Содержит непустую последовательность определений правил трансляций. Данная секция начинается с заголовка, состоящего из двух символов: ‘%%’. Далее, после заголовка, с новой строки идут определения продукций грамматики. Синтаксис определения аналогичен синтаксису приложения Parglare, но с ограничением на имена нетерминалов – они не должны состоять из одних прописных букв, т. к. все имена токенов должны быть полностью в верхнем регистре. Пустое тело правила (обозначающее пустую строку) обозначается как *%empty*;
* Секция эпилога. Данная секция следует после секции правил. Для её отделения от секции правил используется тот же заголовок ‘%%’. Не является обязательной.

Ниже описана грамматика арифметических выражений в виде файла грамматики GNU Bison:

%token NUMBER

%%

s: expr line se;

se: expr line se | %empty;

line: expr ‘\n’;

expr: expr ‘+’ term | expr ‘-‘ term | term ;

term: term ‘\*’ factor | term ‘/’ factor | factor ;

factor: ‘(‘ expr ‘)’ | NUMBER ;

%%

yylex(){

return getNumber();

}

Листинг 2. Грамматика арифметических выражении в нотации GNU Bison.

Одним из главных недостатков данного формата является отсутствие задания лексических правил для терминальных символов грамматики. Они определяются программой *lex*, которая определяет собственный формат описания лексем. Всю работу по сборке последовательности символов в терминалы (в слова) приходится выполнять вручную, с использованием предопределённой функции *yylex*. Она должна вернуть положительное целое число, связанное с определённым токеном. Программа автоматически назначает определённым токенам их коды, так что для возвращения кода достаточно набрать имя токена. Функция также должна сохранить распознанную лексему в переменной *yylval*. В листинге 2 работа функции переложена на функцию *getNumber*, которая собирает число (как целое, так и дробное) в переменную *yylval* и возвращает код токена NUMBER. Ещё одним существенным минусом является отсутствие поддержки операторов группировки и повторения, предоставляемых ANTLR.

Также программа использует имена токенов (терминалов) в верхнем регистре, что опять может привести к ошибкам в определении имён правил грамматики.

Однако программа способна справляться с леворекурсивными грамматиками, в отличие от приложения ANTLR. Она также способна обрабатывать правую рекурсию. Программа способна работать с генератором лексических анализаторов *lex*, позволяя ей определять лексические правила для токенов, задаваемых текущим форматом грамматики.

Соответственно, разрабатываемое программное обеспечение должно обладать следующими преимуществами, перед известными решениями:

1). Принимать грамматику в известном, стандартизированном, удобочитаемым форматом данных. Одним из таких форматов данных является объектная нотация JavaScript – JSON.

2). Давать пользователю информацию о построении анализатора.

3). Предлагать пользователю подавать построенному синтаксическому анализатору текстовые файлы для анализа принадлежности к языку, описываемой грамматикой, на основе который был построен анализатор.

4). Возвращать в качестве результата анализа выходной файл, содержащий промежуточный код исходной программы, или список обнаруженных ошибок в ходе анализа.

**1.4. Входной формат грамматики.**

В качестве входного формата грамматики был выбран формат данных JSON, описанный в [2]. JSON является удобочитаемым и простым форматом данным, который поддерживают большинством приложений и информационных систем.

Контекстно-свободная грамматика состоит из четырёх элементов: множества терминалов *T*, нетерминалов *N*, продукций *P* и стартового символа *S.* JSON-документ, который будет хранить информацию об этих множествах, состоит из JSON-объекта с четыремя свойствами:

* Свойство “terms” представляет множество терминалов грамматики в виде JSON-объекта, содержащего коллекцию пар *ключ–значение. Ключом* является имя терминала, которое должно быть уникальным в пределах объекта. *Значением* является строка, содержащее регулярное выражение. Синтаксис регулярного выражения был взят из [3, с. 124] и дополнен следующими элементами:
  + “+” – положительное замыкание Клини. Означает повторение шаблона 1 или более раз;
  + [a-z] – классы символов. Являются сокращением многократного использования оператора объединения “|”. Например, указанная запись «[a-z]» означает «(a|b|c|…|z)»;
  + “\_” – зарезервированное регулярное выражение, которое обозначает любую строку из одного символа, т. е. любой одиночный символ;
  + “@” – экранирующий символ. Любой одиночный символ, который следует за ним, воспринимается буквально (как операнд). Чтобы написать выражение для строки из символа “@”, достаточно его продублировать – “@@”. Во многих языках программирования в синтаксисе регулярных выражений для аналогичных действий используют символ обратного слеша ”\”. Но поскольку стандартом [3] определено его использование при разборе JSON-документов, в целях совместимости со стандартом для регулярных выражений был определён экранирующий символ собаки “@”;
  + JSON-литерал *null* обозначает регулярное выражение для множества пустых строк.
* Свойство “nonTerms” представляет JSON-массив, состоящий из строк, обозначающих имена нетерминалов грамматики;
* Свойство “start” представляет собой строку, обозначающую имя нетерминала из массива “nonTerms”, который является стартовым символом грамматики;
* Свойство “productions” хранит продукции грамматики в JSON-массиве. Элементами массива являются JSON-объекты с единственным свойством. Имя такого свойства должно присутствовать в массиве nonTerms (т. е. быть нетерминалом грамматики). Значением такого свойства являются либо массив грамматических символов, либо литерал null, либо одиночный грамматический символ, выраженный строкой. Все грамматические символы, составляющие тела продукций, должны быть выражены в качестве строк, обозначающих имена терминалов или не терминалов (т. е. либо являются элементами массива nonTerms, либо свойствами объекта terms). В случае, когда тело продукции состоит из более чем одного грамматического символа, необходимо использовать JSON-массив. В массиве перечисляются строки (имена терминалов или нетерминалов) в определённом порядке. Литерал null можно использовать для обозначения тела продукции, порождающей пустые строки, вместо имени терминала, значение которого в объекте *terms* равно null. Для создания альтернатив к определённому правилу с именем N необходимо создать новый JSON-объект в данном массиве со свойством, имеющим такое же имя N.

Во многих языках программирования зарезервированы специальные ключевые слова. Можно определить дополнительное, необязательное свойство в объекте грамматики с именем *keywords*, значением которого будет массив строк, представляющих ключевые слова. Все перечисленные строки в данном массиве будут считаться терминальными символами грамматики, у которых имя терминального символа совпадает с его шаблоном, т.е. если их определить в объекте “terms” то имена свойств и их значения совпадут.

Представленный выше формат описывает контекстно-свободную грамматику. Однако данный формат не учитывает семантику языка и обработку семантических действий. В частности, он не определяет, что является идентификатором, что является операндом и оператором (в случае операторных грамматик), и какие семантические действия связаны с каждой продукцией. Кроме того, данный формат не описывает зарезервированный список базовых типов, которые могут понадобятся в случае определения языка программирования с системой типов. Поэтому данный формат необходимо дополнить дополнительными секциями.

Для информации о семантике языка можно определить секцию “meta”. Секция “meta” представляет из себя Json-объект и состоит из следующих свойств:

* “commentLine” – свойство, хранящее имя терминального символа (имя свойства из объекта “terms”) определяющий начало однострочного комментария. При встрече такого токена, лексический анализатор должен продолжать чтение символов, игнорируя лексические правила до тех пор, пока он не прочитает символ новой строки. После этого, лексический анализатор может вернуть значение null, поскольку комментарии игнорируются синтаксическим анализатором.
* “mlCommentStart” – свойство, хранящее имя терминального символа, определяюший начало многострочного комментария. Лексический анализатор после прочтения такого токена будет читать символы с входного потока, также игнорируя лексические правила, но до тех пор, пока он не прочтёт строку символов, являющейся значением свойства “mlCommentEnd”.
* “mlCommentEnd” – хранит строку символов, которая обозначает конец многострочного комментария. Не является элементом грамматики. После прочтения указанной строки, лексический анализатор вернёт null
* “id” – хранит имя терминального символа, который обозначает шаблон для идентификаторов. Все распознанные идентификаторы должны пройти семантическую проверку объявлений имён в таблице символов компилятора.
* “operators” – представляет из себя массив строк, которые обозначают имена терминалов и нетерминалов грамматики, являющимися операторами.
* “operands” – аналогично свойству “operators” хранит операнды языка задаваемого грамматикой.
* “scopeStart” – хранит имя терминального символа, обозначающего начало новой области видимости.
* “scopeEnd” – хранит имя термингального символа, обозначающего конец области видимости.
* “aliases” – представляет из себя Json-объект, хранящий коллекцию пар *имя-значение,* где *имя –* имя терминального символа, а *значение –* его псевдоним (новое имя), который присваивается токену в определённых случаях лексическим анализатором. Например, бинарный минус и унарный минус – это два разных оператора и, следовательно, два разных терминала. Но они имеют один и тот же шаблон – знак “-“. К счастью, неоднозначность можно разрещить, определив следующее правило: бинарный минус находится между двумя операндами, а унарный минус – перед операндом. В данном случае, лексический анализатор должен хранить информацию о предыдущем прочитанном токене и о том, что является оператором, а что операндом. И если предыдуший токен был операндом, то текущий минус является бинарным, иначе минус является унарным. В случае бинарного минуса возвращается обычное имя терминала из объекта “terms”. В другом случае вернётся его псевдоним, определённый в “aliases”.
* “types” – представляет из себя массив Json-объектов. Каждый такой объект определяет базовый тип в системе типов входного языка. Объект хранит два свойства:
  + “name”- хранит имя типа в виде строки.
  + “size” – задаёт информацию о выделяемой памяти для типа в виде целого неотрицательного числа.

Выше описанные секции придают контекстно-свободной грамматике дополнительную семантику. Секции “operands”, “operators” и “aliases” используются для разбора операторов, имеющих один и тот же шаблон, но разный контекст. Что является операндом и оператором определяет семантика языка, определяемого грамматикой. С помощью секции “id” можно определить, что является идентификатором в языке. Секции “commentLine”, “mlCommentStart”, “mlCommentEnd” определяют комментарии. Секции “scopeStart” и “scopeEnd” задают границы области видимости имён переменных, определяемых в языках программирования, а секция “types” определяет имена базовых типов языка с системой типов.

Технически, комментарии можно было бы описать в виде отдельных правил контекстно-свободной грамматики. Поскольку они игнорируются синтаксическим анализатором, то их можно размещать в любом месте входной цепочки. Однако, учитывая, что они могут быть в любом месте входной цепочки, то количество правил, такой грамматики будет очень велико. Например, рассмотрим одиночное правило грамматики, состоящее из двух грамматических символов:

Пусть **с –** символ, обозначающий последовательность комментариев. Поскольку его можно размещать в любом месте входной цепочки, то продукция с заголовком *B* пополнится новыми правилами:

Для трёх символьного правила получим:

Очевидно, что в рамках КС-грамматики очень трудно описать комментарии. Секции “mlCommentStart”, “mlCommentEnd”, “commentLine” решают данную проблему.

Теперь, приведём пример входного файла грамматики, использующего выше описанный, дополненный формат. Ниже представлена грамматика булевых и арифметических выражений. Данная грамматика является грамматикой операторного предшествования. Она порождает строки из вложенных блоков, каждый из которых содержит одно или несколько выражений присваивания, разделённых между собой точкой с запятой. Самый первый блок должен кончаться символом точка “.”. Выражение присваивание состоит из имени переменной, символа равно “=” и булевого выражения. Булевое выражение в свою очередь может содержать арифметическое выражение. Символ “-“ в данной грамматике обозначает два терминала: бинарный и унарный минус. Для отличия бинарного минуса, от унарного, для унарного оператора был определён псевдоним с именем *um*, и список операндов. Идентификатор – это терминал с именем *id*, что явно указано в свойстве объекта “meta” с именем “id”. Однострочный комментарий начинается со строки, удовлетворяющей шаблону для терминала “#”, а многострочный комментарий начинается со строки, удовлетворяющей шаблону терминала “/\*”. Многострочный комментарий заканчивается последовательностью символов “\*/”.

{  
 **"terms"**: {  
 **"+"**: **"+"**,

**"dot"**: **"."**,  
 **"-"**: **"-"**,  
 **"\*"**: **"\*"**,  
 **"/"**: **"/"**,  
 **"%"**: **"%"**,  
 **"^"**: **"^"**,  
 **"("**: **"("**,  
 **")"**: **")"**,  
 **"{"**: **"{"**,  
 **"}"**: **"}"**,  
 **"<"**: **"<"**,  
 **"<="**: **"<="**,  
 **"="**: **"="**,  
 **"=="**: **"=="**,  
 **">"**: **">"**,  
 **">="**: **">="**,  
 **"<>"**: **"<>"**,  
 **";"**: **";"**,  
 **"&&"** : **"&&"**,  
 **"||"** : **"@|@|"**,  
 **"~"** : **"~"**,  
 **"->"**: **"->"**,  
 **"#"**: **"#"**,  
 **"/\*"**: **"/@\*"**,  
 **"\*/"**: **"@\*/"**,  
 **"id"**: **"[A-Za-z\_]([A-Za-z\_0-9]+|empty)"**,  
 **"num"**: **"[0-9]+"**,  
 **"realNum"**: **"[0-9]+.[0-9]+(((E|e)(-|empty)[0-9]+)|empty)"**,  
 **"empty"**: **null** },  
 **"keywords"**: [**"true"**,**"false"**,**"not"**],  
 **"meta"**: {  
 **"mlCommentStart"**: **"/\*"**,  
 **"mlCommentEnd"**: **"\*/"**,  
 **"id"**: **"id"**,  
 **"commentLine"**: **"#"**,  
 **"operands"**: [**"id"**,**"num"**,**"realNum"**,**"true"**,**"false"**],  
 **"aliases"**: {  
 **"-"**: **"um"**,  
 **"+"**: **"up"** }  
 },  
 **"nonTerms"**: [**"E"**,**"AE"**,**"T"**,**"F"**,**"PE"**,**"OP"**,**"OPLIST"**,**"P"**,**"IMPL"**,**"CONJ"**,**"DISJ"**,**"UE"**],  
 **"productions"**: [{**"P"**: [**"{"**,**"OPLIST"**,**"}"**,**"dot"**] },  
 {**"OPLIST"**: [**"OPLIST"**,**";"**,**"OP"**]},{**"OPLIST"**: **"OP"**},  
 {**"OP"**: [**"id"**,**"="**,**"IMPL"**]},{**"OP"**: [**"{"**,**"OPLIST"**,**"}"**]},  
 {**"IMPL"**: [**"IMPL"**,**"->"**,**"DISJ"**]},{**"IMPL"**: **"DISJ"**},  
 {**"DISJ"**: [**"DISJ"**,**"||"**,**"CONJ"**]},{**"DISJ"**: **"CONJ"**},  
 {**"CONJ"**: [**"CONJ"**,**"&&"**,**"E"**]},{**"CONJ"**: **"E"**},  
 {**"E"** : [**"AE"**,**"<"**,**"AE"**]},{**"E"** : [**"AE"**,**"<="**,**"AE"**]},{**"E"** : [**"AE"**,**">"**,**"AE"**]}  
 ,{**"E"** : [**"AE"**,**">="**,**"AE"**]},{**"E"** : [**"AE"**,**"=="**,**"AE"**]},{**"E"** : [**"AE"**,**"<>"**,**"AE"**]},{**"E"**: **"AE"**}  
 ,{**"AE"**: [**"T"**]},{**"AE"**: [**"AE"**,**"+"**,**"T"**]},{**"AE"**: [**"AE"**,**"-"**,**"T"**]}  
 ,{**"T"** : [**"F"**]}, {**"T"**: [**"T"**,**"\*"**,**"F"**]},{**"T"**: [**"T"**,**"/"**,**"F"**]},{**"T"**: [**"T"**,**"%"**,**"F"**]}  
 ,{**"F"**: [**"UE"**]},{**"F"**: [**"F"**,**"^"**,**"UE"**]}  
 ,{**"UE"**: [**"~"**,**"UE"**]},{**"UE"**: [**"up"**,**"UE"**]},{**"UE"**: [**"um"**,**"UE"**]},{**"UE"**: **"PE"**}  
 ,{**"PE"**: **"num"**},{**"PE"**: **"id"**},{**"PE"**: **"realNum"**},{**"PE"**: **"true"**},{**"PE"**: **"false"**},{**"PE"**: [**"("**,**"IMPL"**,**")"**]},  
 {**"PE"**: [**"not"**,**"("**,**"IMPL"**,**")"**]}  
 ],  
 **"start"**: **"P"**}

Листинг 3. Грамматика булевых и арифметических выражений в формате JSON.

Отметим, что секции “keywords”, “meta” и всё, что содержится в “meta” указывать необязательно. Они лишь нужны в том случае, если нужно указать дополнительную информацию о семантике входного языка (в особенности, когда известно, что грамматика принадлежит определённому классу).

Обработчик JSON-документов должен обрабатывать файл в вышеописанном формате по следующей последовательности. В первую очередь читается секция “terms”, затем читается секция “keywords”, после неё читается список нетерминалов “nonTerms”. Далее идёт обработка секции “meta” со следующим порядком обработки свойств: “operands”, “operators”, “commentLine”, “mlCommentStart”, “mlCommentEnd”, “id”, “aliases”, “scopeStart”, “scopeEnd”, “types”.

После обработки “meta”, идёт чтение списка “productions”. В последнюю очередь, читается свойство “start”, обозначающее стартовый символ грамматики. Данный символ должен быть нетерминалом.

Генератор синтаксических анализаторов будет порождать два типа распознователей, для двух определённых классов контекстно-свободных грамматик. Первый тип является нисходящим анализатором для класса LL(1)-грамматик, использующего стэк явно (без рекурсии) и таблицу синтаксического анализа. Второй тип является восходящим анализатором работающим по алгоритму “сдвиг”-“свёртка” для грамматик, принадлежащих классу SLR(1).

**1.5. Формат промежуточного кода.**

Существуют несколько форматов промежуточного кода:

* Многоадресный код с явно именуемым результатом (тетрады)
* Многоадресный код с неявно именуемым результатом (триады)
* Обратная (польская) постфиксная запись.

В данной работе используется промежуточное представление в виде линейной последовательности четвёрок. Они представляют из себя трёхадресный код (т.е. код, команды которого оперируют не более чем с тремя адресами) в котором сначала идёт имя команды, затем два её аргумента, и в последнюю очередь имя адреса для хранения результата. Для них легче провести процедуры оптимизации кода. Они имеют следующий формат:

*operator arg1 arg2 result;*

Здесь, *arg1, arg2* и *result* хранят временные имена переменных, сгенерированные компилятором для промежуточных вычислений. Отметим, что унарные операторы не требуют наличия второго аргумента. В этом случае, компилятор будет генерировать имя “:z”, обозначающее отсутствие значения, если команда не интерпритирует значения аргументов по своему.

Для промежуточного кода были определены специальные команды для вызова методов, условного перехода, циклов, а также для обращения к полям класса. Ниже перечислен список команд, составляющих промежуточный код:

1. Команды присваивания вида: *= x type y,* где *x –* значение выражения, стоящего справа от оператора присваивания ‘=’, *type –* тип переменной результата, а *y –* переменная, в которую запишется результат выражения с указанным типом. Эта команда будет создаваться для каждой инструкции присваивания. Использование второго аргумента для имени типа облегчает проверку типов, позволяя использовать данную команду в качестве оператора приведения типов.

2. Команды унарных операторов вида: *op x :z y.* Они представляют собой выражение: *op(x),* где *x –* является операндом. Имя *y* хранит результат выражения. Имя *“:z”* обозначает отсутсвие значения у второго аргумента, так как он игнорируется данной командой.

3.Команда *PUSH\_P x* – сохраняет значение переменной *x* в стэк. Эта команда генерируется при вызовах методов для сохранения контекста текущего метода, т.е. того метода, в теле которого была прочитана инструкция вызова метода. Контекстом текущего метода являются имена переменных, передаваемых по значению к вызываемому методу, а также имена параметров текущего метода, которые не передаются в качестве аргументов вызываемого метода по ссылке.

4. Команда *POP\_P x* - удаляет текущее значение на вершине стэка и кладёт его в переменную с именем *х*.

5. Команда *GETFIELD class field result.* Берёт значение поля с именем *field* класса с именем *class* и сохраняет его в переменную с именем *result.*

6. Команда *PUTFIELD class field exp.* Кладёт значение выражения *exp* в поле *field* класса *class.*

4. Безусловный переход *GOTO L1 L2.* Осуществляет переход к метке *L1* и сохраняет в стэк меток метку *L2,* которая следует сразу же за инструкцией GOTO, и используется для сохранения текущего потока управления программы.

5. Условный переход вида *IFFALSE x L1.* Эта команда приводит к метке *L1* в том случае, если значение *x* ложно. В противном случае, будет выполнена команда, следующая за командой условного перехода *IFFALSE*.

6. Условный переход вида *IFTRUE x L1 L2.* Эта команда осуществляет переход к метке L1 если значение *x* истинно. Если значение *x* ложно, то выполняется переход к метке L2.

7. Вызов процедур и возврат из них реализуется с помощью команд:

* Сохранение контекста с помощью команд PUSH\_P*.*
* *PARAM x :z p*. – передача значение переменной *х* к параметру вызываемого метода с именем *p*.
* *GOTO* *method L* – переход к метке с именем *method*. Метка *L* сохраняется в стэке меток и идёт сразу за данной командой *GOTO*.
* Команда присваивания вида *= x type :res*. Сохраняет значение выражения, следующего за оператором return и хранимого в переменной *х*, в переменную *:res*. Имя данной переменной зарезервировано и используется для хранения возвращаемого значения методов. После этой команды должен осуществляться переход к той метке, которая находится на вершине стэка меток.
* Возврат текущего контекста метода с помощью команд *POP\_P*.

8. Бинарные операторы выражены обычной четвёркой вида: *binop x y t.* Она вычисляет выражение вида *x binop y* и сохраняет его значение в *t.*

**1.6. Исходная грамматика входного языка.**

Ниже дана грамматика входного языка, множество правил которой являются подмножеством правил языка C#. В данном языке в качестве основной единицы компиляции, т.е. программы понимается класс. Программа начинается и заканчивается объявлением класса. Объявление класса состоит из ключевого слова **class**, имени класса, выраженного *идентификатором*, и тела класса. Тело класса представляет из себя блок. Блок состоит из фигурных скобок. В фигурных скобках через точку с запятой перечисляются инструкции. В классе разрешено использовать два вида инструкций: инструкции объявления полей, и методов. Инструкция объявления полей состоит из имени типа и имён полей перечисленных через запятую. Инструкция объявления метода состоит из ключевого слова **def**, имени возвращаемого типа, имени метода и заголовка метода. Заголовок метода содержит в скобках ноль или более формальных параметров. Каждый формальный параметр отделяется от другого запятой. Формальный параметр состоит из имени типа и имени переменной. После скобок идёт тело метода. Тело метода представлено блоком операторов. Блок операторов содержит последовательность операторов. Операторы отделены друг от друга точкой с запятой. Оператором может быть и блок инструкций. Таким образом, язык позволяет определять вложенные блоки. Кроме блока, оператором может быть условный оператор, как полный (содержащий секцию else), так и неполный, причём в обоих случаях, секция if может содержать как одиночный оператор, так и блок. Секция else также содержит либо одиночный оператор, либо блок. Оператором также может быть цикл while. Цикл while состоит из ключевого слова **while,** логического выражения в скобках и тела цикла, представленного либо также одиночным оператором, либо блоком. Оператором также может быть объявление локальной переменной и вызов метода. Наконец, в качестве оператора может быть простое выражение присваивания. Выражение присваивания состоит из имени переменной, в которой будет сохранено значение выражения, оператора присваивания “=” и выражения. Грамматика определяет два вида выражений – логические и арифметические. В качестве литералов, т.е. атомарных единиц выражений может быть целое число, действительное число с дробной частью, отделяемой точкой, имя другой переменной, ключевые слова **true** и **false**, а также одиночный символ, заключённый в кавычки. Для сложных выражений используются круглые скобки. Выражения отрицания интерпретируется как вызов функции “**not**”. Ниже дана таблица приоритета операции при вычислении выражений. Высший приоритет имеет выражение в круглых скобках или вызов метода. Остальные операторы перечислены в таблице в порядке убывания их приоритета.

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Ассоциативность | Значение | Приоритет |
| **not** | правая | Логическое отрицание | 9 |
| **um, up** | правая | Унарный минус, унарный плюс | 8 |
| ^ | правая | Возведение в степень | 7 |
| \*,%,/ | левая | Умножение, деление, | 6 |
| +,- | левая | Сложение, вычитание | 5 |
| >, >=, <, <=, ==, <> | левая | Операции отношения: больше, больше или равно, меньше, меньше или равно, равно, не равно | 4 |
| || | левая | Логическое “ИЛИ” | 3 |
| && | левая | Логическое “И” | 2 |
| -> | левая | Импликация | 1 |

Табл. 4. Приоритет операций в выражениях.

Оператор присвавивания имеет самый низкий приоритет. Для языка определены базовые типы, перечисленные в следующей таблице:

|  |  |  |  |
| --- | --- | --- | --- |
| Имя типа | Память | Является числовым ? | Множество значений |
| int | 4 | да | Целые числа в диапазоне: |
| long | 8 | да | Целые числа в диапазоне |
| float | 4 | да | Числа с плавающей точкой одинарной точности. |
| double | 8 | да | Числа с плавающей точкой двойной точности. |
| char | 2 | да | Одиночная буква, чей числовой код в диапазоне |
| bool | 1 | да |  |
| void | 0 | нет | Пустое множество |

Табл. 5. Базовые типы языка.

Названия базовых типов являются ключевыми словами. Для типа bool определены также два литерала *true* и *false,* обозначающие истинну или ложь логического выражения. Они эквиваленты нулю и единице и тоже являются ключевыми словами. Для типа char определён терминальный символ **ch**, представляющий одиночный символ заключенный в одинарные кавычки. Целое число для типов int и long выражено терминалом с именем **num**. Для чисел типа float и double определён терминал **realNum.**

Для унарного минуса и унарного плюса используются терминалы с именами **um** и **up** соответственно. Эти имена перечислены в секции aliases, поскольку они совпадают с шаблонами бинарного минуса и плюса.

Ниже дана грамматика исходного языка в формате данных JSON, который был описан в настоящей работе ранее.

{  
 **"terms"**: {  
 **"+"**: **"+"**,  
 **"-"**: **"-"**,  
 **"\*"**: **"\*"**,  
 **"/"**: **"/"**,  
 **"%"**: **"%"**,  
 **"^"**: **"^"**,  
 **"("**: **"("**,  
 **")"**: **")"**,  
 **"{"**: **"{"**,  
 **"}"**: **"}"**,  
 **"<"**: **"<"**,  
 **"<="**: **"<="**,  
 **"="**: **"="**,  
 **"=="**: **"=="**,  
 **">"**: **">"**,  
 **">="**: **">="**,  
 **"<>"**: **"<>"**,  
 **";"**: **";"**,  
 **"&&"** : **"&&"**,  
 **"||"** : **"@|@|"**,  
 **"->"**: **"->"**,  
 **"#"**: **"#"**,  
 **","**: **","**,  
 **"/\*"**: **"/@\*"**,  
 **"id"**: **"[A-Za-z@\_]([A-Za-z@\_0-9]+|empty)"**,  
 **"num"**: **"[0-9]+"**,  
 **"realNum"**: **"[0-9]+.[0-9]+(((E|e)(-|empty)[0-9]+)|empty)"**,  
 **"ch"**: **"\'\_\'"**,  
 **"empty"**: **null** },  
 **"keywords"**: [**"class"**,**"true"**,**"false"**,**"if"**,**"else"**,**"while"**,  
 **"int"**,**"long"**,**"float"**,**"double"**,**"char"**,**"bool"**,**"void"**,**"not"**,**"def"**,**"ref"**,**"return"** ],  
 **"meta"**: {  
 **"mlCommentStart"**: **"/\*"**,  
 **"mlCommentEnd"**: **"\*/"**,  
 **"id"**: **"id"**,  
 **"commentLine"**: **"#"**,  
 **"scopeStart"**: **"{"**,  
 **"scopeEnd"**: **"}"**,  
 **"operands"**: [**"id"**,**"num"**,**"realNum"**,**"true"**,**"false"**,**"ch"**],  
 **"scopeTypes"**: [**"CLASS"**,**"METHOD"**,**"BLOCK"**,**"GLOBAL"**],  
 **"operators"**: [**"<"**,**"<="**,**">"**,**">="**,**"=="**,**"<>"**,**"="**,**"+"**,**"-"**,**"\*"**,**"/"**,**"%"**,**"^"**,**"->"**,**"&&"**,**"||"**,**"if"**,**"while"**,**"class"**,**"def"**,**"return"**,**"um"**,**"up"**,**"not"**],  
 **"aliases"**: {  
 **"-"**: **"um"**,  
 **"+"**: **"up"** },  
 **"types"**: [  
 {**"name"**: **"double"**,**"size"**: 8},  
 {**"name"**: **"long"**,**"size"**: 8},  
 {**"name"**: **"float"**, **"size"**: 4},  
 {**"name"**: **"int"**,**"size"**: 4},  
 {**"name"**: **"char"**,**"size"**: 2},  
 {**"name"**: **"bool"**,**"size"**: 1},  
 {**"name"**: **"void"**,**"size"**: 0}  
 ]  
 },  
 **"nonTerms"**: [**"S"**,**"SL"**,**"L"**,**"DL"**,**"D"**,**"MD"**,**"T"**,**"H"**,**"FL"**,**"F"**,**"OPLIST"**,**"OP"**,**"R"**,**"P"**,**"B"**,  
 **"DISJ"**,**"CONJ"**,**"E"**,**"AE"**,**"RELOP"**,**"TE"**,**"FA"**,**"UE"**,**"PE"**,**"OEXP"**,**"CB"**,**"FP"**,**"ELS"**,**"LDL"**,**"CALL"**,**"AP"**,**"AL"**,**"ARG"**,**"EXP"** ],  
 **"productions"**: [{**"S"**: [**"class"**,**"id"**,**"{"**,**"CB"**,**"}"**]},  
 {**"CB"**: **"SL"**},{**"CB"**: **"empty"**},  
 {**"SL"**: [**"SL"**,**"L"**]},{**"SL"**: **"L"**},  
 {**"L"**: [**"DL"**]},{**"L"**: **"MD"**},{**"DL"**: [**"T"**,**"D"**,**";"**]},  
 {**"D"**: [**"D"**,**","**,**"id"**]},{**"D"**: **"id"**},{**"T"**: **"int"**},{**"T"**: **"long"**},{**"T"**: **"float"**},{**"T"**: **"double"**},{**"T"**: **"char"**},  
 {**"T"**: **"bool"**},{**"T"**: **"void"**},{**"T"**: **"id"**},  
 {**"MD"**: [**"def"**,**"T"**,**"id"**,**"H"**]},{**"H"**: [**"("**,**"FP"**,**")"**,**";"**]},{**"H"**: [**"("**,**"FP"**,**")"**,**"R"**]},  
 {**"OP"**: [**"return"**,**"OEXP"**]},  
 {**"OEXP"**: **"EXP"**},{**"OEXP"**: **"empty"**},  
 {**"FP"**: **"FL"**},{**"FP"**: **"empty"**},  
 {**"FL"**: [**"FL"**,**","**,**"F"**]},{**"FL"**: **"F"**},  
 {**"F"**: [**"T"**,**"id"**]},{**"F"**: [**"ref"**,**"T"**,**"id"**]},{**"P"**: [**"OP"**,**";"**]},{**"P"**: **"R"**},  
 {**"R"**: [**"{"**,**"OPLIST"**,**"}"**]},{**"OPLIST"**: [**"OPLIST"**,**"P"**]},{**"OPLIST"**: [**"P"**]},  
 {**"P"**: [**"if"**,**"("**,**"B"**,**")"**,**"P"**,**"ELS"**]},{**"ELS"**: [**"else"**,**"P"**]},{**"ELS"**: **null**},  
 {**"P"**: [**"while"**,**"("**,**"B"**,**")"**,**"P"**]},  
 {**"OP"**: [**"id"**,**"="**,**"EXP"**]},{**"OP"**: **"LDL"**},{**"LDL"**: [**"T"**,**"D"**]},{**"OP"**: **"EXP"**},  
 {**"RELOP"**: **"<"**},{**"RELOP"**: **"<="**},{**"RELOP"**: **">"**},{**"RELOP"**: **">="**},{**"RELOP"**: **"=="**},{**"RELOP"**: **"<>"**},  
 {**"EXP"**: **"B"**},{**"EXP"**: **"CALL"**},  
 {**"CALL"**: [**"id"**,**"("**,**"AP"**,**")"**]},{**"AP"**: **"AL"**},{**"AP"**: **"empty"**},  
 {**"AL"**: [**"AL"**,**","**,**"ARG"**]},{**"AL"**: **"ARG"**},{**"ARG"**: **"EXP"**},  
 {**"B"**: [**"B"**,**"->"**,**"DISJ"**]},{**"B"**: **"DISJ"**},  
 {**"DISJ"**: [**"DISJ"**,**"||"**,**"CONJ"**]},{**"DISJ"**: **"CONJ"**},  
 {**"CONJ"**: [**"CONJ"**,**"&&"**,**"E"**]},{**"CONJ"**: **"E"**},  
 {**"E"**: [**"AE"**,**"RELOP"**,**"AE"**]},{**"E"**: **"AE"**},  
 {**"AE"**: [**"AE"**,**"+"**,**"TE"**]},{**"AE"**: [**"AE"**,**"-"**,**"TE"**]},{**"AE"**: **"TE"**},  
 {**"TE"**: [**"TE"**,**"\*"**,**"FA"**]},{**"TE"**: [**"TE"**,**"/"**,**"FA"**]},{**"TE"**: [**"TE"**,**"%"**,**"FA"**]},{**"TE"**: **"FA"**},  
 {**"FA"**: [**"FA"**,**"^"**,**"UE"**]},{**"FA"**: **"UE"**},  
 {**"UE"**: [**"um"**,**"UE"**]},{**"UE"**: [**"up"**,**"UE"**]},{**"UE"**: **"PE"**},  
 {**"PE"**: **"id"**},{**"PE"**: **"num"**},{**"PE"**: **"realNum"**},{**"PE"**: **"true"**},{**"PE"**: **"false"**},{**"PE"**: **"ch"**},  
 {**"PE"**: [**"("**,**"B"**,**")"**]},{**"PE"**: [**"not"**,**"("**,**"B"**,**")"**]}  
 ],  
 **"start"**: **"S"**}

Листинг 6. Описание языка и его грамматики в виде JSON.

Из выше представленного описания, терминальными символами грамматики будут следующее множество: {**id, true, false, class, def, num, realNum, ch, ',', ; , (, ), {, }, if, else, while, um, up, +, -, \*,/, %,^,&&, ||, ->, <, <=, >, >=, ==, <>, = , int, long, float ,double, bool, char, void,return}.** Множество ключевых слов является подмножеством множеств терминалов и состоит из следующих элементов: {**int, long, float, double, char, bool, void, ref, true, false, class, def, return, if, else, while, not** }. Множество нетерминалов грамматики будет следующее множество: *N* = *{S,CB, SL, L, DL,D, MD, T, H, FP,FL, F, OEXP, EXP, CALL, AP, AL, ARG, OPLIST, P, OP,LDL, ELS, R, B ,DISJ, CONJ, E, AE, RELOP, TE, FA, UE, PE}*. Стартовым символом грамматики является нетерминал S. Множество правил грамматики дано ниже, начиная со стартового правила.

Листинг 7. Грамматика исходного языка.

Нетерминалы B, DISJ, CONJ, E, AE, TE, FA, UE, PE выражают приоритет операций (логических и арифметических). Нетерминал RELOP группирует операции сравнения. Нетерминал DL представляет объявления полей класса. Нетерминал P представляет инструкции (пер. с англ. statements). Инструкции представляют из себя либо один оператор с точкой запятой, либо инструкцию if-else, либо цикл while, либо блок со списком инструкций. Нетерминал OPLIST описывает последовательность инструкций, выраженных нетерминалом P. Отметим, что не всякая инструкция является оператором, хотя все операторы являются инструкциями (если за ними следует точка с запятой). Инструкция if-else не является оператором, поскольку она выражена непосредственно нетерминалом *P*. Операторы выражены нетерминалом OP. Таким образом, список операторов может быть выражен в терминах списка инструкций, если каждый оператор будет оканчиваться символом точка с запятой. Нетерминал SL выражает список объявлений полей или методов. Синтаксис для объявления локальных переменных внутри метода и объявления переменных экземпляра класса одинаков, хотя для них используются разные нетерминалы. Для полей используется нетерминал DL, а для локальных переменных – нетерминал LDL. Всё дело в правилах вида: и . Так как правило *DL* оканчивается символом точка с запятой, то если его сделать оператором (т.е. добавить правило вида OP -> DL), то получится дублирование символа точки с запятой. Поэтому нетерминал DL не обозначает правило для оператора, а для решения проблемы дублирования был добавлен новый нетерминал LDL, который содержит все грамматические символы правила DL, кроме последнего символа, т.е. точки с запятой. Сам нетерминал LDL сворачивается в нетерминал OP, позволяя таким образом решить неоднозначность с локальными переменными и полями объекта класса.Нетерминал OEXP используются для генерации как пустых так и не пустых выражений, поскольку за оператором return может как быть, так и не быть выражения. Тип *void* обозначает метод, который не должен возвращать никакого выражения. Соответственно нетерминал EXP обозначает непустое выражение. Непустое выражени представляет из себя либо вызов метода, представленный нетерминалом CALL, либо нетерминал B, выражающий корень логического выражения, так как согласно приоритету операции, сначала вычисляется логическое выражение, а затем арифметическое. Корнем арифметических выражений является нетерминал AE.

Отметим, что данная грамматика не является LL(1) грамматикой из-за наличия левой рекурсии. Её нельзя привести к LL(1) грамматике. Даже после устранения левой рекурсии и удаления общих префиксов у любых двух альтернатив каждой продукции она снова не будет принадлежать классу LL(1) из-за наличия следующих правил:

Данные продукции не содержат левой рекурсии. Кроме того, продукция для нетерминала ELS не содержит альтернативы с общим префиксом. Следовательно, правила останутся неизменны после устранения левой рекурсии и обших префиксов альтернатив из грамматики. Множества FIRST для данных правил, без учёта остальных правил грамматики будут равными:

Однако, согласно правилам вычисления множества FOLLOW все элементы множества FOLLOW(P) будут добавлены во множество FOLLOW(ELS), аналогично, все элементы из множества FOLLOW(ELS) будут лежать во множестве FOLLOW(P). Это происходит из-за того, что и нетерминал ELS и нетерминал P являются крайними правыми символами продукций. Поскольку за нетерминалом P лежит нетерминал ELS, и FIRST(ELS) содержит символ пустой строки *empty,* то во множество FOLLOW(P) добавится элемент *else.* Так как FOLLOW(ELS) содержит все элементы из множества FOLLOW(P), то он будет также содержать *else.* В итоге получим, что множества FIRST(ELS) и множества FOLLOW(ELS) пересекаются, что противоречит определению LL(1)-грамматик.

Можно попытаться изменить грамматику так, чтобы она принадлежала классу LL(1) грамматик и порождала тот же язык, что и исходная грамматика. Однако в общем случае оказывается невозможным сделать подобные преобразования без изменения исходного языка. Можно рассмотреть более широкий класс грамматик и проверить, принадлежит ли данная грамматика из листинга 6, к данному классу. В данном случае, грамматика принадлежит к классу грамматик SLR(1). Доказательство данного факта дано в разделе 2.6, а также в приложении 1. Отметим, что в исходной грамматике для выражения списка элементов грамматических символов (список параметров, имён переменных, инструкций) используется левая рекурсия. Она является более оптимальным решением, чем правая рекурсия, поскольку при левой рекурсии, синтаксический анализатор попытается провести свёртку символов в стэке как можно раньше. При правой рекурсии, анализатор проведёт свёртку, только после прочтения всех элементов списка. Таким образом, при левой рекурсии размер стэка увеличивается на постоянную велечину, равную длине строки грамматических символов, составляющей тело продукции. А при правой рекурсии стэк растёт динамически, в зависимости от последовательности входных символов. Более подробный разбор рекурсивных правил дан в [4].

**2. Конструкторский раздел.**

**2.1. Основная структура программы.**

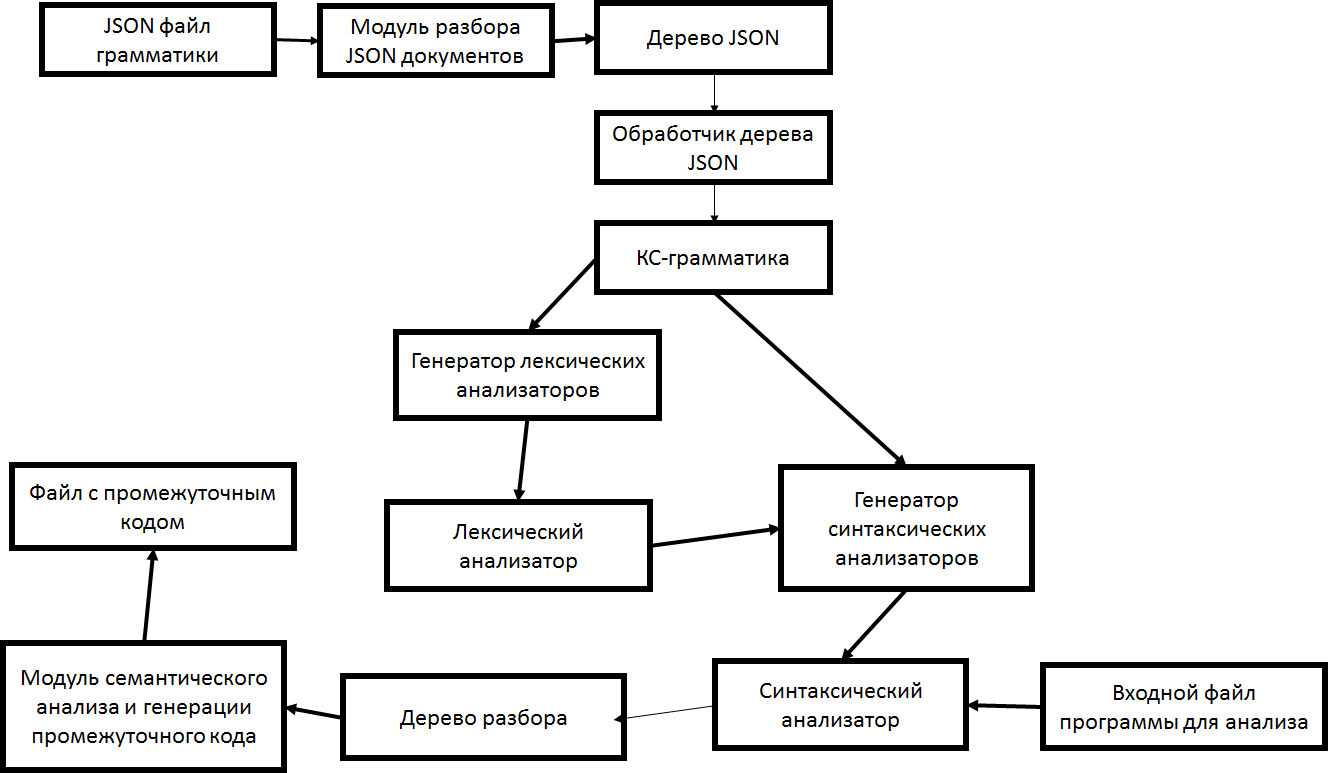
Основные компоненты разработанной программы приведены на рис. 1. Модуль разбора JSON документов читает файлы в формате, определённом в разделе 1.4. Модуль возвращает фактическую структуру документа, представленной в виде дерева. Модуль обработки дерева JSON извлекает из данного дерева информацию, необходимую для построения грамматики. В случае успеха, модуль возвращает экземпляр класса Grammar, представляющего информацию о контекстно-свободной грамматике. Данная информация, кроме множеств терминалов, нетерминалов, стартового символа и множества продукции, может содержать дополнительные данные (о имени терминала, обозначающего идентификатор, шаблоны для однострочных и многострочных комментариев и т.д.). Дополнительные данные содержатся в отдельном объекте класса GrammarMetaInfo. Экземпляр класса Grammar подаётся сначала модулю генератора лексических анализаторов, для построения лексического анализатора, затем, вместе с лексическим анализатором, грамматика подаётся на вход генератору синтаксических анализаторов. Модуль возвращает синтаксический анализатор. Данный анализатор, принимает на вход текстовые файлы, для анализа и возвращает дерево разбора. Синтаксический анализатор сохраняет информацию об идентификаторах в таблице символов. Организация таблицы символов дана в разделе 2.4. Дерево разбора затем подаётся на вход модуля семантического анализа и генерации промежуточного кода. Модуль возвращает в итоге текстовый файл, содержащий последовательность четвёрок, каждая из которых разделена символом новой строки. 

Рис. 8. Основные компоненты транслятора.

**2.2. Модуль разбора JSON-документов.**

Так как за основу формата входных данных грамматики был взят формат обмена информации, выраженной в объектной нотации JavaScript (JavaScript Object Notation, JSON), то для разбора JSON-документов был разработан вспомогательный модуль. Данный модуль возвращает типа *JsonObject*, представляющего дерево JSON-документа. Корнем такого дерева является объект типа *JsonObject*. Листьями являются объекты, представляющие элементарные скалярные значения. Они выражены типами *JsonString*, *JsonBoolean*, *JsonNull*, *JsonNumber*, *JsonRealNumber*, для строк, логических значений (true, false), для литерала null, и для чисел соответственно. Промежуточными узлами являются объекты типа *JsonArray* и *JsonObject*, представляющие массивы и вложенные объекты. Все выше перечисленные типы имеют общего родителя – базового типа *JsonElement*<T>, представляющего элементы json документа и определяющего метод *getValue()*, который возвращает фактическое значение элемента. Для отображения свойств в виде строк вида *“имя\_свойства” : значение*, реализация наследуется от типа *LinkedHashMap*<*String*,V> типом *StrLinkedHashMap*<*String*, V>, переопределяя метод *toString()*. Для получения значения определённого свойства внутри объекта используется метод экземпляра типа *JsonObject* – *getProperty*. Чтобы получить значение, используя нотацию *json-path*, в JsonObject определён метод экземпляра *getElement*. Оба метода принимают в качестве параметра строку. *Json-path* представляет из себя последовательность имён свойств, отделённых точками. Первая точка указывает на корневой объект, т.е. на сам экземпляр типа JsonObject. Вторая и последующие точки ссылаются на вложенные объекты. Например, пусть JsonObject определяет свойство *meta* со значением *JsonObject*, который содержит свойство *id* со значением *a.* Тогда, чтобы получить данное значение, необходимо вызвать метод *getElement*, передав ему следующую строку ***“.meta.id”***. Тип *SimpleJsonParser* реализует модуль разбора, предоставляя два метода:

* *parse(filename : String) : JsonObject* – читает файл с именем filename и возвращает объект JsonObject при успешном разборе json документа, иначе вернёт *null*.
* *parse(stream: InputStream) : JsonObject* – делает тоже самое, что и предыдуший метод, но читает не из файла, а из потока данных.

Для обработки чисел в различном формате был определён вспомогательный тип *ProcessExp*. Он предоставляет следующий метод:

* *parse(num : String, exp : String, etype : char, b : int, sign : int) : double* – преобразовать строковое представление числа *num*, которое содержит экспоненту с указанным числом в виде строки *exp* и представленное в системе счисления, указанной в переменной *b*, и имеющий положительный или отрицательный знак (переменная *sign* хранит два значения для плюса и минуса – это 1 и -1). Переменная *etype* хранит символ, указывающий на основание экспоненты. Он может быть одним из следующих символов:
  + *h* или *H* – основание равно 10.
  + *b* или *B* – основание равно 2.
  + *e* или *E* - основание равно 10.

Состояния парсера *SimpleJsonParser* заданы перечислением *JsParserState*.

Ниже дана диаграмма классов для всех типов модуля разбора JSON документов.

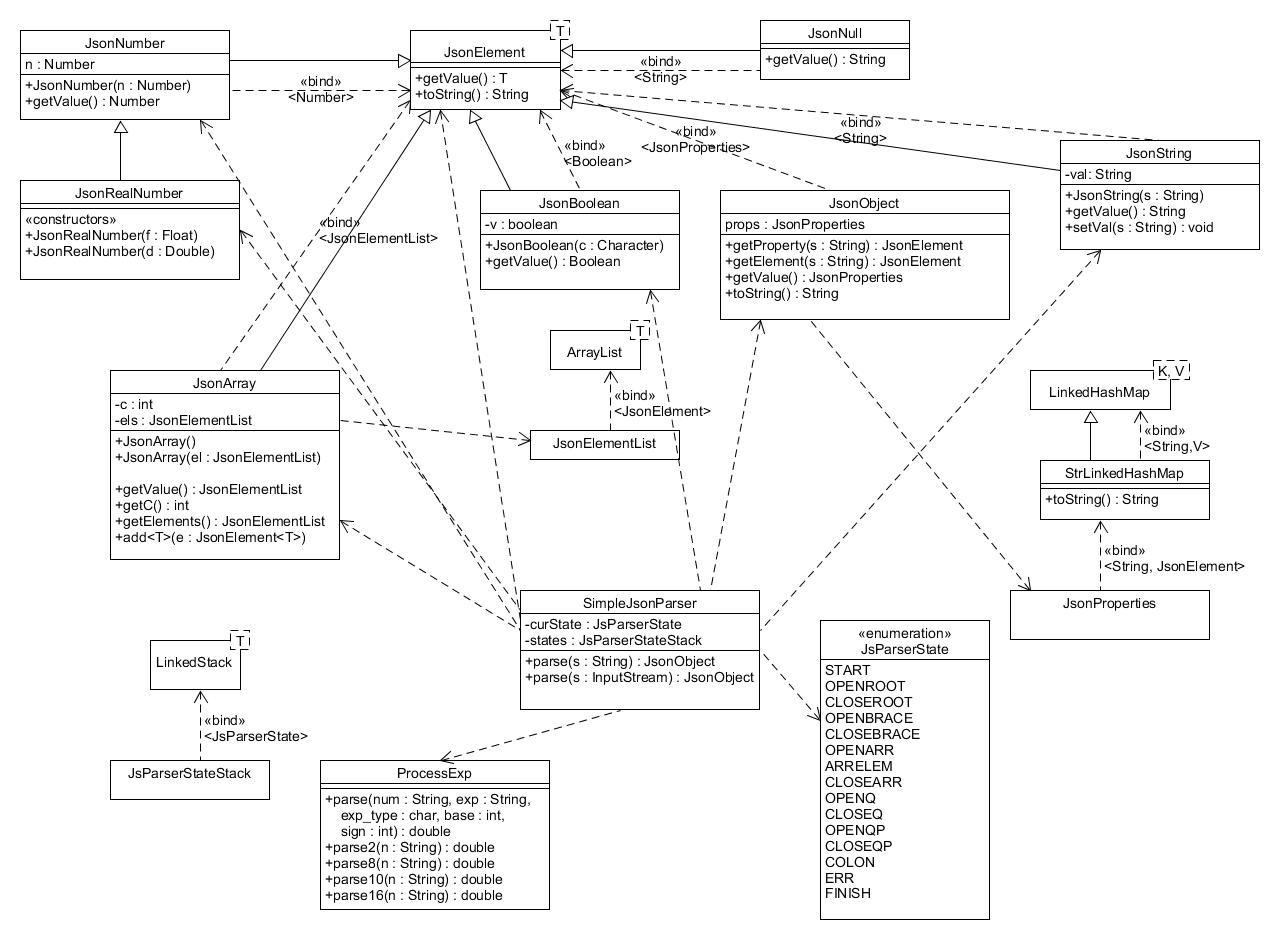


Рис. 9. Диаграмма классов модуля разбора JSON документов.

**2.3. Модуль преобразования JSON-дерева в объект контекстно-свободной грамматики.**

Данный модуль извлекает информацию, полученную из файла в формате json и хранимой в объекте типа *JsonObject*, возвращаемый парсером *SimpleJsonParser*. Для доступа к свойствам *JsonObject* используются методы *getElement* и *getProperty*. Модуль пытается найти и прочитать свойства “terms”, “nonTerms”,”productions”,”start”. Если хотя бы одно из этих свойств не определено, модуль возвращает ошибку, в виде экземпляра класса *InvalidJsonGrammarException*. Что касается секций “meta” и “keywords”, то данные секции не обязательны. Модуль сможет их прочитать если они были определены корректно, в соответствии с требованиями в разделе 1.4. Что касается секций многострочных комментариев: секция “mlCommentStart” требует, чтобы была также определена секция “mlCommentEnd”. И наконец, при обнаружении наличии той или иной секции, из тех, что были определены в разделе 1.4., модуль проверяет тип значения, которые они содержат. Например, если секция “keywords” была определена, но не как массив, а как объект, то модуль просто её не прочитает. А если секция “terms” была определена как-то иначе, чем объект с последовательностью пар *имя:значение*, то модуль вернёт сообщение об ошибочном формате грамматики. Функционал модуля представлен конструктором класса *Grammar*. Обработка информации из JSON-документов осуществляется в конструкторе класса, принимающего на вход корень Json-дерева в виде объекта типа *JsonObject*. Данный класс содержит также методы преобразования грамматик. Методы экземпляра класса Grammar - *deleteUselessSymbols, deleteNonReachableSymbols, getNonEmptyWordsGrammar, getNonCycledGrammar,* позволяют получить новые грамматики, эквивалентные исходной, но не содержащие бесполезных символов, пустых правил, или циклов. Статический метод класса Grammar *deleteLeftRecursion* позволяет устранить левую рекурсию из грамматики. Данный метод сначала вызывает процедуры *deleteUselessSymbols*, *deleteNonReachableSymbols*, *getNonEmptyWordsGrammar* и *getNonCycledGrammar* для получения грамматики без бесполезных символов, цепных правил и пустых правил. Затем она вызывает процедуру *getIndexedGrammar*, которая упрощает упорядочивание нетерминалов в алгоритме. Метод *getIndexedGrammar* возвращает новую грамматику, присваивая новые имена для каждого заголовка продукции (нетерминалов). Новое имя состоит из старого имени и порядкового номера, которое отделяется символом нижнего подчёркивания. Номера присваиваются в порядке появления в выходной цепочке (вывод). Например, пусть дана грамматика арифметических выражений со следующим множеством правил и стартовым символом *E*:

Тогда метод присвоит следующие имена заголовкам продукций:

Процедура *deleteLeftRecursion* основана на 3-фазном методе устранения рекурсии [5]. Метод *deleteLeftFactor* проводит левую факторизацию грамматики. Данный класс также содержит экземпляр типа *GrammarMetaInfo*, который хранит семантическую информацию о входном языке. Класс *Grammar* также предоставляет методы доступа для получения значения полей объекта типа *GrammarMetaInfo*. Класс *Grammar* имеет следующие поля:

* *T : Set<String>* - множество имён терминалов.
* *N : Set<String>* - множество имён нетерминалов.
* *lex\_rules Map<String,List<String>>* - множество лексических правил грамматики. Отображение имени терминала во множество строк регулярного выражения. В данной реализации список содержит не более чем одного элемента.
* *S : String* – стартовый символ грамматики
* *E : String* – имя терминала, обозначающего пустую строку.
* *P : Map<String,Set<GrammarString>>* - множество продукций грамматики.
* *meta : GrammarMetaInfo* – представляет объект, хранящий семантику входного языка.

Объект типа GrammarMetaInfo хранит семантику в своих полях, перечисленных ниже:

* *keywords : Set<String>* - множество ключевых слов языка.
* *operands : Set<String>* - множество операндов языка.
* *operators : Set<String>* - множество операторов языка.
* *aliases : Map<String,String>* - множество псевдонимов для терминальных символов.
* *commentLine : String* – имя терминала, обозначающего начало однострочного комментария.
* *mlStart : String* – имя терминала, обозначающего начало многострочного комментария.
* *mlEnd : String* – строка символов, обозначающая конец многострочного комментария.
* *id : String* – имя терминала, представляющего шаблон для идентификатора. В таблице символов хранятся элементы, ключом которых является лексема идентификатора.
* *begin : String* – имя терминала, обозначающего начало новой области видимости.
* *end : String* – имя терминала, обозначаюшего конец текущей области видимости.
* types : Set<Pair<String,Integer>> - множество пар вида (*имя типа, количество памяти*).

Каждая продукция грамматики имеет тело, выраженное типом *GrammarString*. Тип *GrammarString* хранит список грамматических символов. Каждый грамматический символ представлен типом *GrammarSymbol,* имеющий следующие поля:

* *val : String* – имя символа (которое содержится во множествах терминалов или нетерминалов).
* *type : char* – тип символа, нетерминал или терминал. Тип имеет следующие значения:
  + ‘t’ – терминал
  + ‘n’ – нетерминал
  + ‘e’ – ошибка.

В следующей диаграмме перечислены все типы данного модуля приложения.

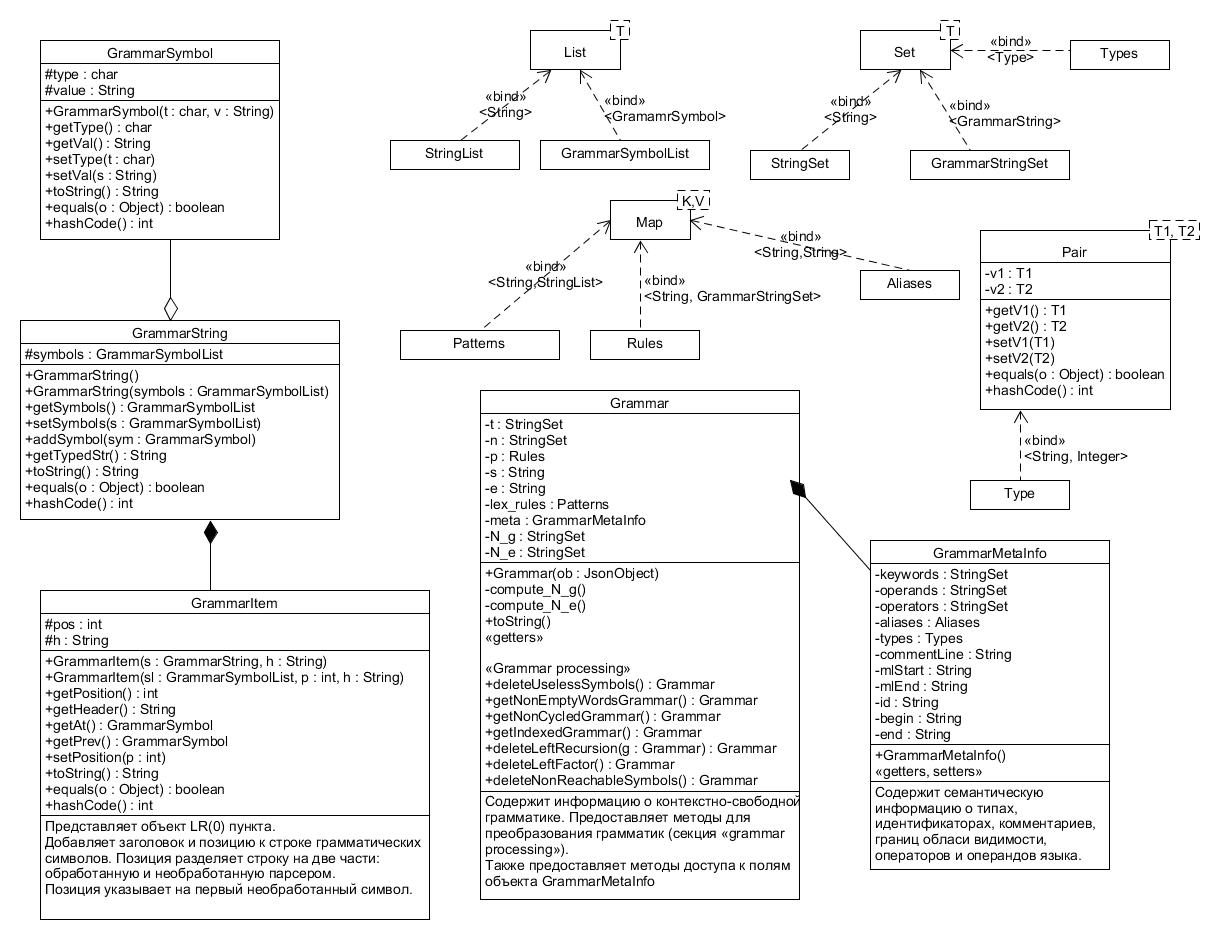
****

Рис.10. Диаграмма классов модуля преобразования JsonObject в объект типа Grammar, а также модуля обработки объектов типа Grammar.

**2.4 Модуль генератора лексического анализатора.**

Данный модуль строит лексический анализатор на основе модели детерминированного автомата, используемой в таких программах, как Flex, JFlex.

Построение детерминированного автомата состоит из пяти фаз.

1) Для каждого терминального символа, обрабатывается его шаблон. Шаблон представляет из себя регулярное выражение, описанное в параграфе 1.4, с использованием имени терминального символа, для обозначения шаблона пустых строк. В каждом шаблоне, имя такого терминального символа, заменяется на специальный символ, обозначающий пустую строку.

2) Шаблон записывается в обратной польской записи. Сначала к шаблону приписывается оператор конкатенации “^” между операндами и раскрываются классы символов. Данную работу выполняет метод класса *RegexInputParser* *addConcat(s : String, regex: RegexRPNParser)*, принимающий строку регулярного выражения *s* и экземпляр класса преобразователя в обратную польскую запись *RegexRPNParser.* Классы символов представляют собой краткую запись выражения объединения двух или более регулярных выражений, и имеют следующий вид: [A-Z023d-e]. В данном примере определён диапазон символов от A до Z включительно, а также символы 0, 2,3, и диапазон символов от d до e включительно. Классы символов были определены, чтобы пользователь мог использовать более сокращённую запись выражений. Далее, выражение с операторами конкатенации и раскрытыми классами символов подаётся в качестве аргумента методу *getInput(s : String)* экземпляра типа *RegexRPNParser.* Он принимает строку *s* регулярного выражения, и преобразует её в обратную польскую запись. Данная запись возвращается в виде стэка символов. С помощью этой записи и алгоритма конструктора Томпсона строится недетерминированный конечный автомат (НКА). Данный алгоритм реализован в виде процедуры класса *RegexInputParser: buildNFA(expr : LinkedStack<Character>, parser: RegexRPNParser, id : Elem<Integer>).* Данный метод принимает стэк символов *expr*, экземпляр класса *RegexRPNParser* *parser*, и текущее значение идентификатора *id*, представленный объектом типа *Elem<Integer>,* для вершины графа. Метод возвращает объект типа *CNFA*, представляющего недетерминированный конечный автомат (далее НКА), со множеством допускающих состояний. Данный тип расширяет тип *NFA*, который представляет НКА, но только с одним допускающим состоянием. Тип *NFA* в свою очередь наследуется от типа *Graph*, который представляет структуру графа. Класс Graph хранит список вершин и рёбер графа, а также позволяет добавлять новую вершину, рекурсивно добавляя также связанные (т.е. смежные) с ней другие вершины и рёбра, инцидентные ей. Вершина представлена типом *Vertex* имеющая следующие поля:

* *name : String* – имя вершины. Используется для графического изображения графа.
* *val : String* – метка вершины, её значение. Хранит имя терминала, для которого определено регулярное выражение. Используется лексическим анализатором для построения объектов типа *Token*.
* *isStart : boolean* – является ли вершина стартовой в конечном автомате.
* *isFinish : boolean* – является ли вершина принимающей в конечном автомате.
* *isDead : boolean* – является ли вершина тупиковой.
* *edges : List<Edge>* - список рёбер, индидентных данной вершине.

3) Все построенные НКА объединяются в комбинированный НКА. Комбинированный НКА представлен типом *CNFA*. Он содержит несколько допускающих состояний, причём каждое допускающее состояние имеет метку, со значением имени терминала, которому соответствует регулярное выражение. Данная информация используется при построении детерминированнго конечного автомата (далее ДКА), который представлен классом *DFA*. Данный класс наследуется от класса графа *Graph*. Ребро графа представлено типом *Edge* и оно имеет два поля типа *Vertex*, для узлов графа. Ребро является ориентированным, и хранит узлы в разных полях, причём порядок узлов используется при построении автомата. Вся работу по обработке, преобразованию выражения, построению НКА и объекта типа *CNFA* возлагается на тип *FALexerGenerator*. Он предоставляет единственный метод, объединяющий предыдущие 3 фазы:

* *buildNFA(g : Grammar) : CNFA* – принимает на вход объект грамматики *g*, возвращает экземпляр класса *CNFA*, который представляет из себя объединение всех НКА для каждого шаблона терминального символа указанной грамматики, определённого в свойстве “*terms*” объекта *JsonObject*.

4) Полученный комбинированный недетерминированный конечный автомат преобразуется в детерминированный. Для преобразования из НКА в ДКА используется алгоритм подмножеств [6, с. 206], функции *e-closure* и *move* класса *NFA*, а также информация о метках допускающих состояний. Конструктор класса *DFA*, имеющий единственный параметр типа *CNFA*, возвращает объект типа DFA из CNFA. Он выполняет данную фазу.

5) Полученный ДКА минимизируется. Для минимизации используется алгоритм Хопкрофта [7]. Также в целях минимизации удаляется тупиковое состояние автомата. Минимизацию осуществляют две функции класса *DFA*: *minimize* и *makeRecord*. Первая возвращает группы неразличимых состояний. А вторая заполняет новую таблицу переходов для нового ДКА с минимальным числом состояний.

Построенный автомат передаётся как параметр конструктора *DFALexer*. Тип *DFALexer* наследуется от типа *DFA* и реализует интерфейс *ILexer*. Интерфейс *ILexer* предоставляет метод для синтаксического анализатора, чтобы он смог читать токены по одному. Метод имеет следующее описание:

*recognize(stream: InputStream) : Token* – распознаёт одну лексему из входного потока данных типа *InputStream* и возвращает токен, выраженный типом *Token*. Если данная лексема представляет из-себя начало однострочного или многострочного комментария, то будет возвращено значение *null*, так как синтаксический анализатор игнорирует токены.

Другой метод *setGrammarMetaInfo(meta : GrammarMetaInfo),* передаёт объект *meta*, хранящий семантическую информацию входного языка, лексическому анализатору. Лексический анализатор должен в состоянии отличать распознанные лексемы от ключевых слов и комментариев, а также отличать унарный оператор от бинарного на основе прочитанных лексем и информацией о том, что является операндом, а что оператором.

Метод *reset*() перезапускает лексический анализатор, и вызывается лишь после процедуры синтаксического анализа, т.е. после того, как будет прочитан целый файл. Данный метод обнуляет счётчик текущей строки и столбца, а также указатель на поток символов, хранимый в типе *LookaheadBufferedLexer*.

Класс *LookaheadBufferedLexer* реализует интерфейс *ILexerIO*, который предоставляет работу с источником входных данных. Интерфейс *ILexerIO*, предоставляет следующие методы:

* *getch(s : InputStream) : int* – проверяет, не пуст ли буфер, и если буфер не пустой, то удаляет с него текущий символ и возвращает его целочисленный код. Иначе метод прочитает ровно один символ с входного потока, переместит указатель на следующий символ входного потока. Если поле объекта *LookaheadBufferedLexer* для указателя текущего входного потока не установлено, то данный метод также сохранит экземпляр типа InputStream *s* в данное поле.
* *getFilech(s : InputStream) : int* – читает символы входного потока без проверки буфера.
* *ungetch(c : char) : int* – сохраняет символ *с* в буфер. При переполнении буфера генерирует ошибку. Максимальный размер буфера установлен по умолчанию равным 256 символам.

Методы экземпляра класса *LookaheadBufferedLexer* используется в классе DFALexer для чтения из файла. Тип *DFALexer* использует жадную стратегию при распознавании лексем. Т.е. он читает самый длинный корректный префикс входной строки. При распознавании лексем, *DFALexer* проверяет, была ли распознана лексема для идентификатора, и если она также содержится в списке ключевых слов, то вернуть токен, представляющее ключевое слово. Тип *Token*, представляет токены, которые читает синтаксический анализатор. Данный тип имеет три поля: *name, lexeme, type*. Поле *name* хранит имя терминала грамматики, поле *lexeme* – лексему входного потока, поле *type* – тип символа в грамматике: терминал, нетерминал, ошибка. Для ключевых слов поле *name* совпадает по значению с полем *lexeme*. Для идентификаторов, поле *name* имеет имя того терминального символа, который был выбран в качестве идентификатора (свойство “id” объекта “meta”). Если лексема представляет из себя начало однострочного комментария, то анализатор пропускает символы входного потока до тех пор, пока не прочитает либо символ новой строки либо символ конца входного потока. В обоих случаях, анализатор останавливается читать, запоминает текущий символ, и возвращает *null* синтаксическому анализатору, оповещая его о том, что никакой лексемы найдено не было. Синтаксический анализатор в этом случае попытается получить следующую лексему из входного потока. В случае многострочного комментария, чтение продолжается либо до конца входного потока, либо пока анализатор не прочитает последовательность символов, сигнализирующих о том, что был прочитан многострочный комментарий. Аналогично с однострочным, лексический анализатор вернёт *null*. Для тех терминальных символов, у которых были определены псевдонимы, анализатор вернёт псевдонимы, но только в том случае, если были определены и предыдущий токен не являлся операндом. Для унарных операторов здесь используется префиксная нотация, в которой оператор стоит слева от операнда. В случае ошибки чтения файла, возвращается исключение типа *IOException*.

Ниже даны диаграммы всех типов, необходимых для построения генератора лексических анализаторов. Типы *Elem<T>, Pair<T1,T2>* - служат для передачи одиночного аргумента или пары аргументов по ссылке. Тип *RegexInputParser* делает предобработку строки, представляющей регулярное выражение, которую затем передаёт типу *RegexRPNParser*, который возвращает стэк символов, представляющий обратную польскую запись выражения.

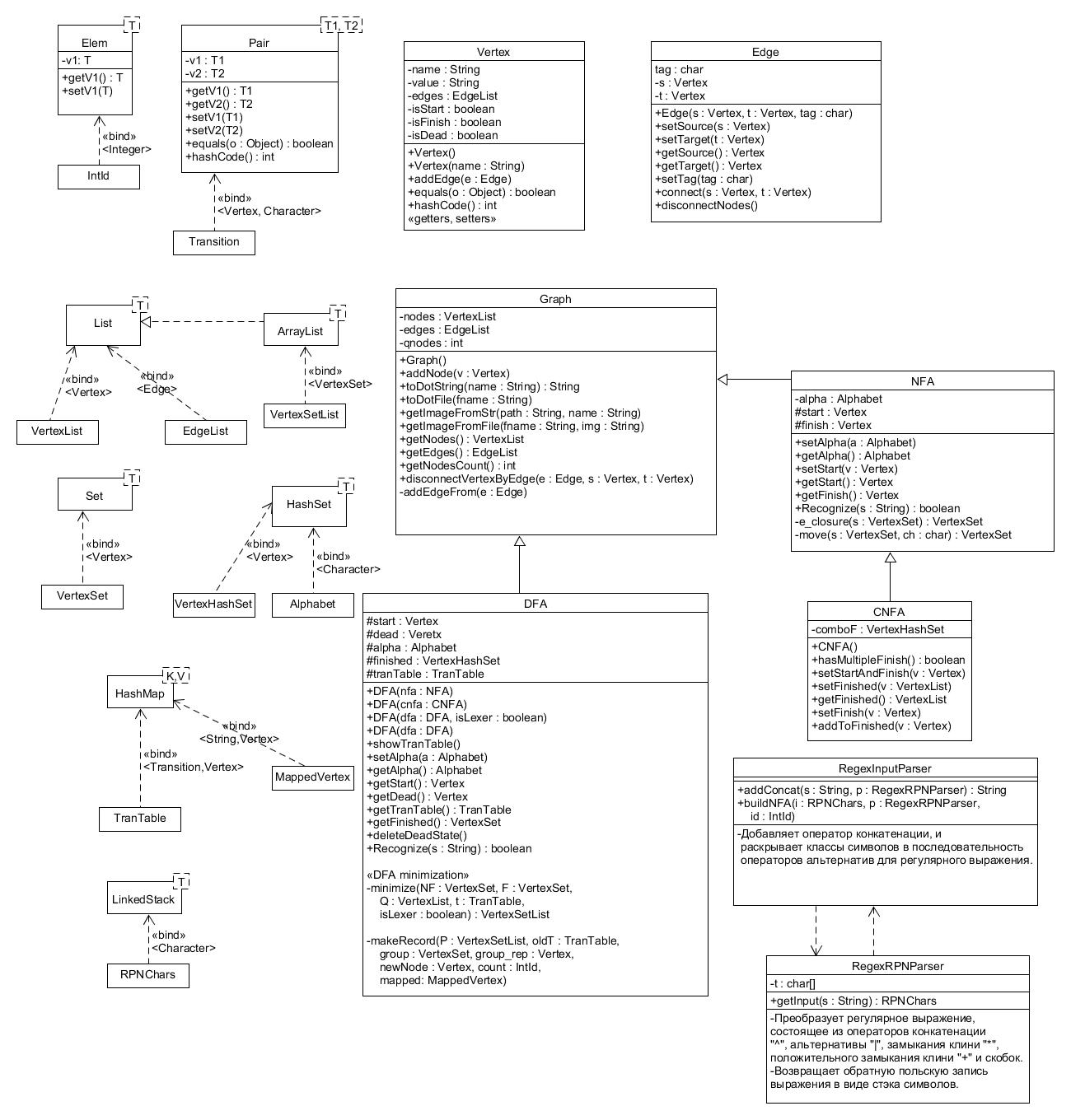


Рис. 11. Диаграмма классов для структур графа и автоматов.

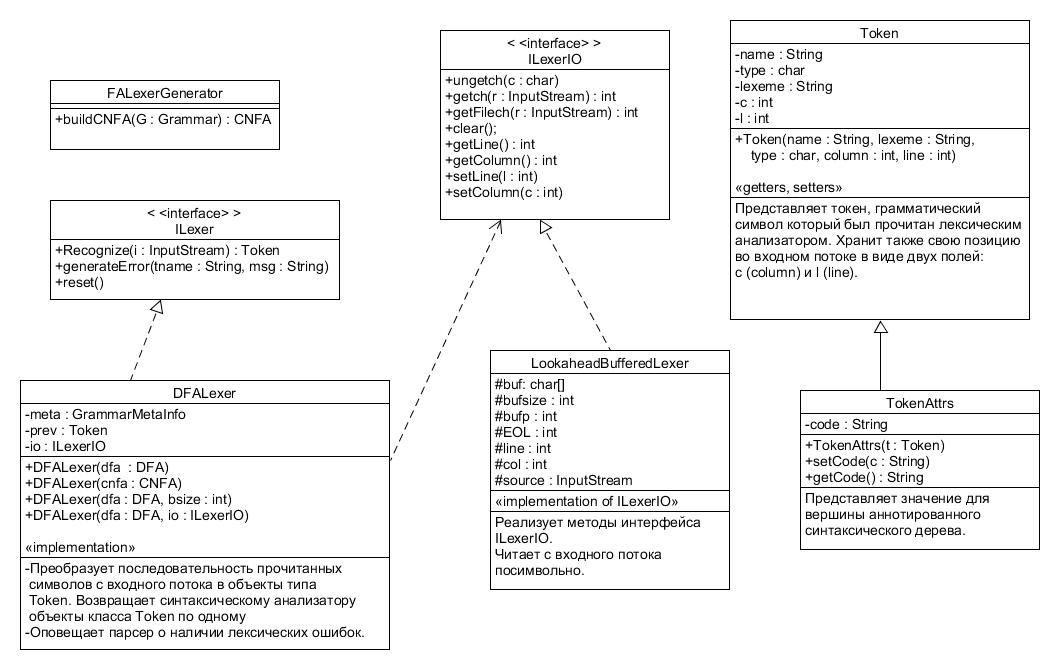


Рис. 12. Диаграмма типов генератора лексических анализаторов.

**2.5 Организация таблицы символов и модуль обработки деревьев.**

Компилятор активно взаимодействует с таблицей символов, операции поиска и вставки элемента в таблицу выполняются очень часто. Поэтому был выбран один из самых эффективных и быстрых методов организации таблицы – хэширование. Время на вставку или поиск элемента зависит лишь от функции хэширования. Поскольку количество идентификаторов в программах может быть сколь угодно велико, то не исключены коллизии. Существуют две основные тактики разрешения коллизии – во-первых, это повторный поиск свободного места, т.е. повторное хеширование с учётом определённой схемы. Например, если для идентификатора A, хеш-функция h(A) вернула занятую ячейку таблицы, то вычислить также адрес i-ой ячейки с помощью выражения h\_i (A) = (h(A) + pi) mod N, где N – максимальное значение хэш функции, а pi – случайное число из последовательности случайных чисел, , с количеством элементов, равных N. Вторая тактика – это использование метода цепочек. В данной работе используется комбинированный метод с использованием промежуточной хэш-таблицы. Значением хэш-функции является индекс ячейки в таблице. Содержимое ячейки таблицы является указателем на корень дерева двоичного поиска. Данная структура позволяет упорядочивать данные для сокращения времени поиска. Изначально все ячейки хэш-таблицы пусты. При встрече нового идентификатора вычисляется хэш-функция. Если она возвращает пустую ячейку, то создаётся дерево двоичного поиска, корнем дерева является новая запись в таблице символов, и в пустую ячейку записывается ссылка на корень дерева. Если ячейка занята, то элемент вставляется в двоичное дерево поиска. Запись, хранимая в таблице символов представлена типом *SInfo*, хранящий имя, которое однозначно идентифицирует запись в дереве, а также ссылку на саму запись, представленную экземпляром типа *Entry*. *SInfo* реализует интерфейс *Value<T>*, предоставляя метод *getValue() : String*, который возвращает лексему для идентификатора. Тип *Entry* хранит четыре атрибута для идентификатора: это сам идентификатор, его тип, его категория и количество байт памяти, выделяемой для данной переменной. Для данного класса, определены четыре следующих подкласса:

* *ClassInfo* – тип записи для классов. Хранит список записей типов *MethodInfo* и *FieldInfo*, представляющих методы класса и его поля.
* MethodInfo – тип записи для методов класса. Хранит в поле *params* список объектов типа *ParameterInfo*.
* *FieldInfo* – тип записи для полей класса.
* *ParameterInfo* – тип записи для параметров метода. Дополнительно хранит булевое поле *isRef*, обозначающее передачу параметра по ссылке или по значению.

Сама таблица символов представлена типом *STable*. Она имеет конструктор с двумя параметрами, которые задают диапазон значений хэш-функций. Два крайних значения, переданных в конструктор, также принадлежат к области значений хэш функций. Таким образом, в конструкторе указываются границы отрезка значений хэш функций. Таблица также предоставляет четыре основных метода:

* *init()* – инициализирует хэш-таблицу пустыми ячейками.
* *clear()* – стирает все записи в ячейках хэш-таблицы.
* *add(s : SInfo)* – добавляет объект типа *SInfo* с соответсвующей записью типа Entry в таблицу. В качестве ключа таблицы используется значение хэш функции от строки, хранимой в поле объекта типа *SInfo* с именем *sName*. В дереве, в качестве ключа используется фактическое значение строки, т.е. без хеш-функции.
* *get(s* : String) : *SInfo* – получить запись таблицы по ключу. Параметр *s* представляет лексему для идентификатора, который хранится в таблице символов.

Для входного языка определена блоковая область видимости, с вложенными блоками. Фигурные скобки задают границу области видимости. Синтаксические правила грамматики налагают на блоки дополнительные ограничения. Например, блок класса, который сразу следует за его именем, позволяет определять внутри себя либо переменные, т.е. поля класса, либо методы класса. Согласно грамматике языка, запрещено объявлять вложенные блоки внутри блока класса. Но внутри блока метода (тела метода) разрешается объявлять сколь угодно много вложенных блоков. Однако, внутри метода запрещено объявлять другие методы. Кроме того, внутри класса и метода запрещено объявлять также вложенные классы и методы. В свою очередь, данное обстоятельство приводит к тому, что одних фигурных скобок, обозначающих границы области видимости, оказывается недостаточным для поиска объявлений переменных, методов и классов. Необходимо также отслеживать контекст блока, т.е. к чему он относится. Например, блок, следующий за именем класса, относится к телу класса, а блок, который следует за списком формальных параметров метода, относится к телу метода. Сам же класс объявляется в глобальной области видимости. Поскольку ключевые слова зарезервированны, то они также объявляются в глобальном контексте. Отметим, что в языке C# имя параметра может быть любым. В свою очередь, оно может перекрывать имя поля класса или имя метода. Поэтому список параметров создаёт свою область видимости, перед областью видимости тела метода. Перечислим же основные области видимости в порядке возрастающей вложенности: от наименее вложенного до наиболее вложенного блока.

1. Глобальная область видимости (GLOBAL). В ней объявлены имена ключевых слов, а также имена классов.
2. Область видимости класса (CLASS). В ней объявлены имена полей класса и его методов.
3. Область видимости параметров метода (PARAMETER). В ней объявлены имена параметров, для определённого метода.
4. Область видимости тела метода (LOCAL). В ней объявлены имена локальных переменных а также все имена переменных, встречающихся во вложенных блоках.

Для семантического анализатора определено перечисление *EntryCategory*, которая обозначает контекст текущего имени, а именно: что обозначает данное имя – класс, поле класса, метод, переменная, параметр. Ниже дано описание значений *EntryCategory*:

* GLOBAL – глобальная область видимости.
* CLASS – обозначает, что было прочитано ключевое слово **class,** за которым идёт имя класса.
* FIELD – область видимости класса.
* METHOD – обозначает, что было прочитано ключевое слово **def,** за которым идёт имя типа возвращаемого значения метода.
* PARAMETER – область видимости параметров метода. Вложена в область видимости класса. Содержит имена параметров метода.
* VAR – область видимости тела метода, а также его вложенных блоков. Содержит имена локальных переменных, которых можно определять в теле метода, а также во вложенных блоках.
* METHOD\_TYPE – обозначает, что данная вершина дерева содержит запись с типом возвращаемого значения метода.
* EXPR\_TYPE – обозначает, что данная вершина дерева содержит запись с типом выражения, которое стоит справа от оператора присваивания.

Имена литералов CLASS и METHOD не относятся ни к одному блоку. Скорее они относятся к ключевым словам **class** и **def**. Причина, по которой были определены два вспомогательных имён CLASS и METHOD кроется в правилах вычисления типа для объявляемых переменных. Для вычисления типа переменных используется первый встречный идентификатор в цепочке обхода дерева или имя базового типа. Здесь CLASS означает отдельное состояние для семантического анализатора, говорящее о том, следующий идентификатор является именем класса, а не именем типа переменных, т.к. было прочитано ключевое слово **class**, а правилом грамматики указано явно, что за ним следует идентификатор. Имя METHOD и ключевое слово **def** используются для различимости имён полей класса и имён методов класса, т.к. при обходе дерева получится такая же цепочка терминальных символов, какая бы получилась при объявлении одиночного поля класса. При чтении параметров метода не возникает никаких проблем с вычислением типов для переменных-параметров. Ниже дана диаграмма классов, формирующих основную структуру таблицы символов.

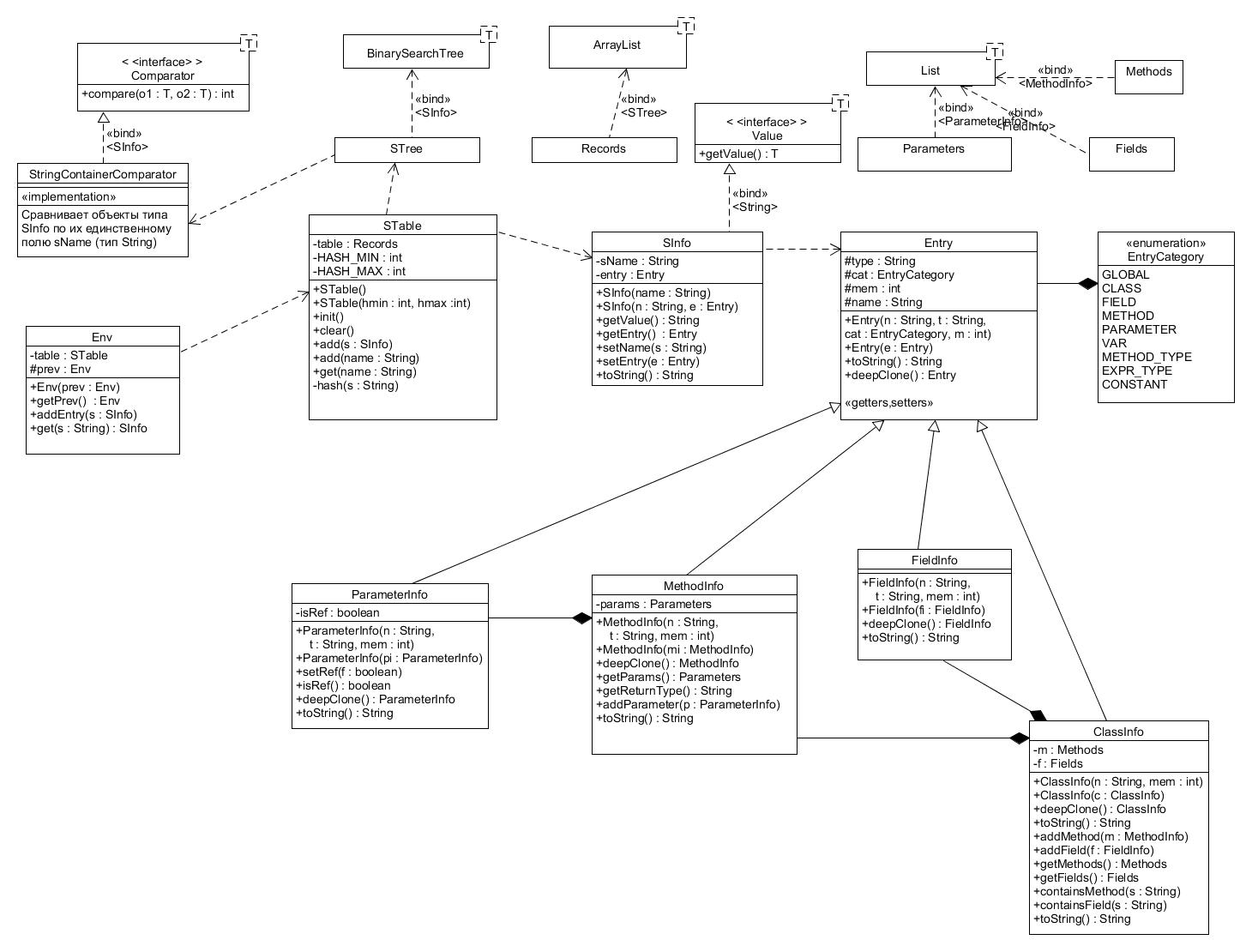


Рис. 13. Диаграмма классов для таблицы символов компилятора.

Двоичное дерево поиска представлено классом *BinarySearchTree*<T>, хранящий элементы типа *SInfo* в лексикографическом порядке имён. Тип *SInfo* хранит лексему идентификатора в поле *sName*. Узлы дерева двоичного поиска представлены типом *LinkedBinaryNode*<T>. Дерево разбора, которое создаёт синтаксический анализатор представлен типом *LinkedTree<T>*, имеющий узлы типа *LinkedNode<T>*. Классы *LinkedNode<T>* и *LinkedBinaryNode<T>* наследуются от класса *Node<T>*, определяющий вершину дерева. Класс *Node<T>* хранит значение типа T (содержимое узла дерева) в поле *val*, и имеет целочисленный идентификатор, хранимый в поле *idx*. Данное поле облегчает процедуру идентификации узла в дереве. Идентификация узла необходима при обходе дерева, так как поскольку некоторые вершины должны быть посещены только один раз, то необходимо отслеживать список посещённых вершин. Из-за того, что возможны случаи, когда при обходе дерева, некоторые вершины должны быть посещены только раз, а другие могут посещаться многократно, то наличие логического поля *isVisited* для каждой вершины дерева не решает проблему идентификации. Дополнительно класс *Node<T>* определяет поле *record*, хранящий ссылку на запись в таблице символов, а именно, на экземпляр класса *Entry*. Класс *LinkedNode<T>* расширяет тип *Node<T>*, предоставляя два дополнительных поля *parent* и *children*, которые хранят родителя узла и список его сыновей. Все они имеют тип *LinkedNode<T*>. Класс *LinkedBinaryNode<T>* предоставляет три поля, хранящих родителя (*parent*), левого и правого сына узла (*left* и *right*), а также набор булевых полей, которые дают следующую информацию: имеет ли вершина только левого сына, или только правого, или она является листом, т.е. не имеет сыновей. Оба типа, *BinarySearchTree<T>* и *LinkedTree<T>* реализуют интерфейс *Tree<T>,* который используется объектом посетителя деревьев. В приложении определены два интерфейса посетителя – *Visitor<T> и SubVisitor<T>,* реализуемыми типами *NRVisitor<T>* и *NRSubVisitor<T>* соответственно. Интерфейс *Visitor<T>* предоставляет следующие методы для обхода дерева. Все они принимают в качестве первого аргумента объект, реализующий интерфейс *Tree<T>*, который является деревом для обхода.

* *preOrder(t : Tree<T>, act : Action<Node<T>>)* – посетить дерево *t* в прямом порядке обхода.
* *inOrder(t : Tree<T>, act : Action<Node<T>>)* – посетить дерево *t* в симметричном порядке.
* *postOrder(t : Tree<T>, act : Action<Node<T>>)* – посетить дерево *t* в обратном порядке.

Интерфейс *SubVisitor<T>* расширяет интерфейс *Visitor<T>,* предоставляя дополнительно три метода для обхода поддеревьев указанного дерева. Они имеют дополнительный третий параметр, имеющий тип *Node<T>*, представляющий корень поддерева. Все методы имеют такое же имя и обходят дерево также. Отличие заключается в том, что обход дерева они начинают не с его корня, а с его определённой вершины, которая передаётся в качестве третьего аргумента данным методам.

Посетители посещают дерево в прямом, обратном или симметричном порядке, и для каждой посещённой вершины дерева выполняют действие. Действие представлено функциональным интерфейсом *Action<T>,* который имеет один параметр – объект типа T. Для посетителей тип параметра метода интерфейса *Action<T>* является типом *Node<T>*, представляющий текущую вершину дерева, которую посетил посетитель. Метод данного интерфейса ничего не возвращает.

Для вызова процедуры обхода дерева, в классах *LinkedTree<T>* и *BinarySearchTree<T>*

Ниже даны диаграммы классов для структур деревьев и таблиц символов.

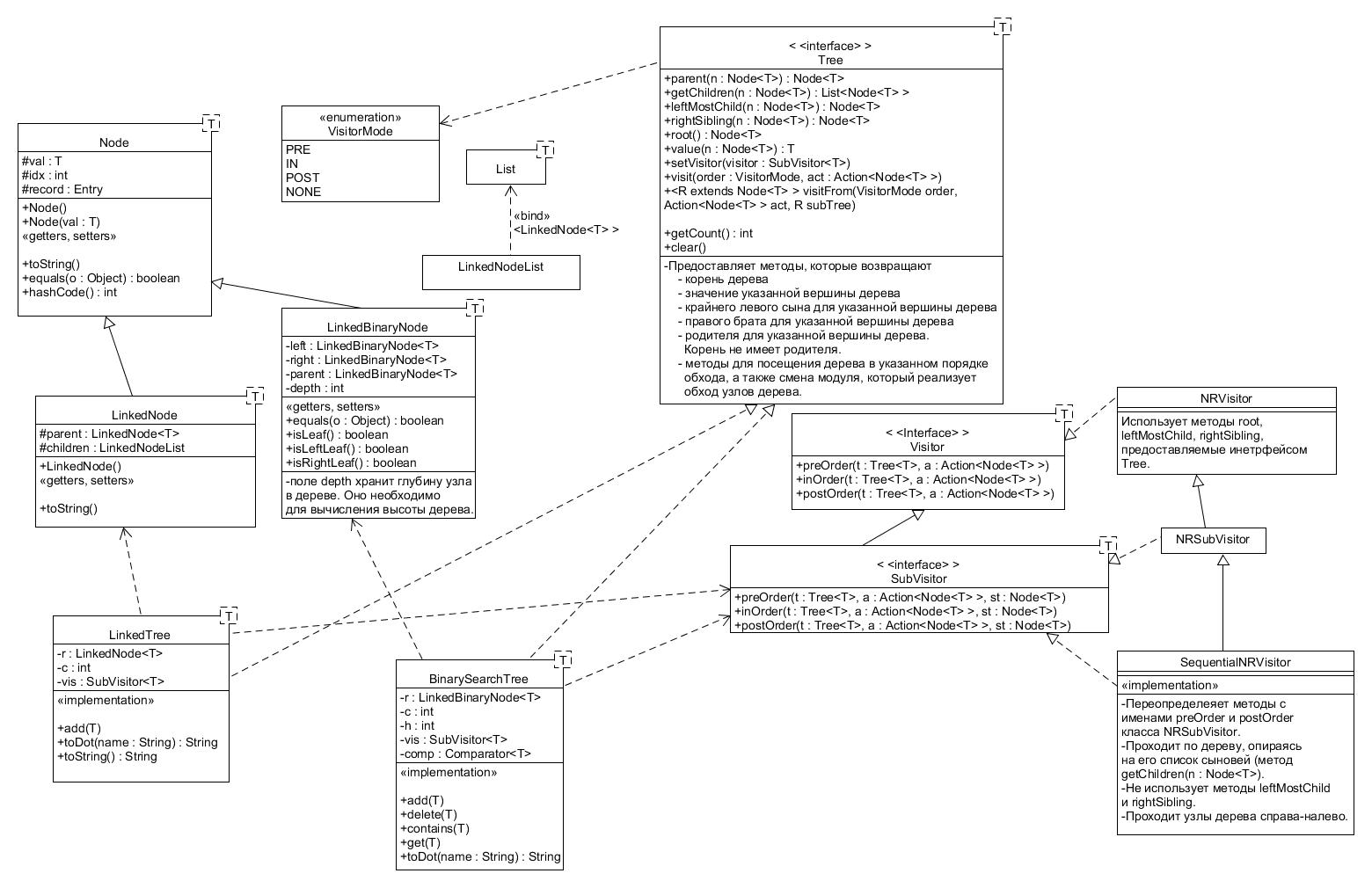


Рис. 14. Диаграмма классов для структур деревьев.

**2.6 Модуль синтаксического анализа.**

В данном модуле определен генератор анализаторов для грамматик класса SLR(1). Данный анализатор является восходящим. Парсер строит дерево разбора снизу вверх, выполняя свёртки и перенос символов в стэк.

Для создания таблицы синтаксического анализа SLR-(1) грамматики, необходимо вычислить функции FIRST и FOLLOW. В классе *LLParserGenerator* определена процедура *firstTable*, возвращающая таблицу значений функции FIRST для одиночного грамматического символа (терминала или нетерминала). Получив данную таблицу, легко написать функцию *first*, вычисляющее множество FIRST, для произвольной строки грамматических символов. В данном классе, эта функция принимает строку из грамматических символов, и построенную таблицу значений функций FIRST для одиночного символа, а также имя терминала, обозначающий пустую строку. Для каждого нетерминала грамматики необходимо вычислить значение функции FOLLOW. Метод *followTable* выполняет данную задачу. Для реализации механизма наследования в случае, когда все элементы из множества FOLLOW(A) для нетерминала А, представляющего заголовок продукции, идут во FOLLOW(B), где нетерминал В – либо является самым крайним правым символом продукции, либо множество *first(Xk…Xn)*, символов грамматики, стоящих справа от B, содержит символ пустой строки, используется шаблон издатель – подписчик. Здесь, подписчиками и издателями являются сами множества FOLLOW. При добавлении элементов из FOLLOW(A) в FOLLOW(B), множество А подписывает подписчика, являющегося множеством В. Кроме того, множество А добавляет все элементы во множество В. В дальнейшем, когда во множество А добавятся новые элементы, эти элементы также добавятся во все множества, которые *подписались* на А. Здесь А – издатель, а В – подписчик. В данном случае В необходимо получать уведомления о добавлении новых элементов во множество А, и в такой ситуации, получить их от А. С этой целью был создан класс *ObservableHashSet*, предоставляющий методы регистрации подписчика. Он хранит всех своих подписчиков в виде списка *observers*, элементы которого представлены тем же типом. Множество FIRST вычисляется для нелеворекурсивной грамматики. Множество FOLLOW вычисляется для исходной грамматики, а не для преобразованной грамматики без левой рекурсии. Построив множества FIRST, FOLLOW, можно построить таблицу синтаксического анализа. Для восходящего LR анализа необходимо построить таблицу ACTION и GOTO, определяющие действия (свёртка, перенос) и состояния анализатора, которые обобщают информацию о содержимом стэка парсера. В случае алгоритма SLR(1) необходимо сначала построить канонический набор множеств LR(0) – пунктов (пер. с англ. LR(0)-items) [6, c. 311]. Каждое такое множество представляет отдельное состояние анализатора. Объект пункта представлен типом *GrammarItem,* который расширяет тип *GrammarString*, предоставляя два дополнительных поля:

* *pos : int* – позиция текущего рассматриваемого символа.
* *header : String* – заголовок продукции.

Данный класс определяет два конструктора:

* *GrammarItem(sym : List<GrammarSymbol>, h : header)* – строит пункт с правилом, заголовок которого имеет имя *h*, и тело, состоящее из списка *sym*. Позиция текущего символа в таком пункте равна первому символу в теле продукции. Данный пункт содержит точку (позицию) с левого края.
* *GrammarItem(sym : List<GrammarSymbol>, h : header, p : int)* – в отличие от предыдущего конструктора, позволяет указать позицию текущего рассматриваемого символа.

Дополнительно в данном классе, кроме методов доступа к полям, определён метод *getAt(),* возвращающий грамматический символ, находящийся на позиции *pos* в списке грамматических символов грамматической строки (тела продукции).

В классе *SLRParserGenerator* определены методы, для построения канонического набора множеств пунктов, а также вычисления функции GOTO.

* *closure(g : Grammar, I : GrammarItem) : Set<GrammarItem>* - возвращает замыкание от множества пунктов, состоящих ровно из одного пункта *I*. Она основана на методе из [6, c. 314].
* *gotoSet(g : Grammar, I : Set<GrammarItem>, x : GrammarSymbol) : Set<GrammarItem>* - для заданного множества пунктов *I*, и грамматического символа *x*, строит новое множество пунктов, в котором каждый пункт получается перемещением точки на одну позицию вправо, если эта точка указывает на символ *х*. Для каждого такого нового пункта, функция добавляет в результирующее множество его замыкание (вызов функции *closure*).Функция *gotoSet* основана на функции из [6, c. 315].
* *buildLRAutomaton(g : Grammar) : LR\_0\_Automaton* – строит канонический набор множеств LR(0)-пунктов для заданной грамматики и вычисляет значения функции GOTO. Возвращает объект типа *LR\_0\_Automaton*, который представляет структуру синтаксической таблицы со значениями функций ACTION и GOTO, а также канонического набора множеств C, и значений функций FIRST и FOLLOW. Перед началом вычислений, грамматика расширяется, путём добавления нового стартового символа и следующей продукции: . Также, после добавления нового стартового символа, вычисляется множество FIRST для каждого грамматического символа грамматики. Затем данное множество сохраняется в объекте типа *LR\_0\_Automaton*. Т.к. из-за устранения левой рекурсии, нетерминалы имеют новые имена с порядковым номером, то в конструкторе класса *LR\_0\_Automaton* при передаче значений функции FIRST, происходит замена ключей в таблице FIRST, т.е. новые имена заменяются на старые (имена нетерминалов в исходной, леворекурсивной грамматике). Алгоритм построения канонического набора множеств пунктов дан в [6, c. 316].

Вызвав процедуру *buildLRAutomaton* и получив объект типа *LR\_0\_Automaton*, остаётся вычислить значения функций ACTION и FOLLOW.

В конструкторе класса *LR\_0\_Automaton* вычисляется значения функции FOLLOW с помощью метода *followTable* класса *LLParserGenerator*. Там же осуществляется вызов процедуры *initActions,* которая заполняет таблицу синтаксического анализа, вычисляя ACTION. Заполнение таблицы происходит по алгоритму [6, c. 323]. После заполнения таблицы, объект передаётся в конструктор класса *LRParser*. *LRParser* – тип, реализующий алгоритм синтаксического анализа [6, c. 320]. Он наследуется от абстрактного типа Parser, который настраивает передаваемый через конструктор лексический анализатор, и сохраняет его в наследуемом поле *lexer*. Он также сохраняет имя терминала для шаблона пустой строки в поле *empty*, хранит флаг *isParsed*,определяющий было ли построено дерево разбора, и задаёт поле *mode*, хранящий значение из перечисления *ParserMode*. Перечисление *ParserMode* позволяет настроить вывод сообщений о работе парсера следующим образом:

* DEBUG – показать все шаги работы анализатора.
* ERR – показать шаги с сообщениями об ошибках, а также предупреждений компилятора.
* WARNING – показать только шаги с сообщениями о предупреждениях.
* HIDE – не выводить информацию о работе парсера.

С помощью метода экземпляра класса *LRParser* *parse(s : String)* при успешном разборе возвращается объект типа *LinkedTree<Token>* представляющий дерево разбора, вершины которого хранят объекты типа *Token*.

Ниже дана диаграмма типов модуля синтаксического анализа.

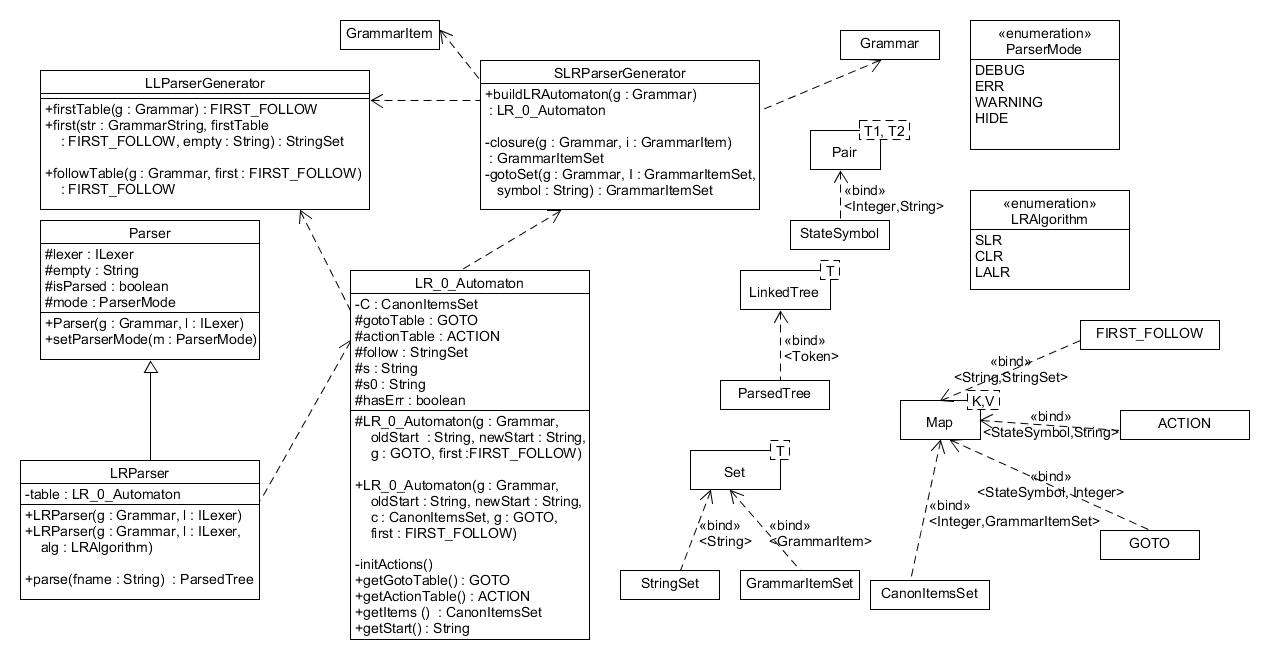


Рис. 15. Диаграмма класов модуля синтаксического анализа.

**2.7 Модуль семантического анализа и генерации кода.**

За семантический анализ и генерацию кода отвечает тип *SemanticAnalyzer*. Данный тип осуществляет семантические проверки объявлений переменных и типов выражений. Он реализует интерфейс *Action<Node<Token>>*. Класс *SemanticAnalyzer* хранит ссылку на объект грамматики и абстрактного синтаксического дерева. Также в классе определены два поля-счётчика, выражающих количество временных переменных и меток, а также два поля флага, которые используются в ходе проверок объявлений переменных: *isExpr* и *isRef*. Они определяют, явялется ли встреченный идентификатор частью выражения или параметром, который передаётся по ссылке.Кроме этого, класс также определяет набор полей для сохранения текущего контекста области видимости, а именно:

При проверке типов, он также генерирует промежуточный код для выражений и вызовов процедур (методов класса), также вставляя операторы приведения типов (команды для операторов присваивания, описанных в разделе 1.5). В нём определены следующие процедуры:

* *hasErrors() : boolean* – определяет наличие или отсутствие семантических ошибок.
* *showErrors()* – напечатать список всех семантических ошибок.
* *openScope()* – переключиться к новой области видимости. Переход осуществляется при достижении границы новой области видимости, выраженной именем терминального символа, представляющего значение поля *begin* объекта *GrammarMetaInfo*.
* *closeScope()* – выйти из текущей области видимости. Выход осуществляется при достижении границы области видимости, выраженной именем терминального символа, представляющего значение поля *end* объекта *GrammarMetaInfo*.
* *search(n : LinkedNode<Token>)* – ищет все выражения объявления переменных и заносит информацию в таблицу символов. Также сохраняет ссылку на запись в вершине дерева, хранящее имя переменной (метода, параметра или класса).
* *typeCheck(n : LinkedNode<Token>)* – осуществляет проверку типов локальных переменных, выражений и параметров методов. Проверяет, соответствует ли число фактических параметров метода с числом переданных аргументов. Она проходится по дереву в прямом порядке, и для определённых вершин, обходит поддеревья с указанными вершинами в качестве корней в обратном порядке, вызывая метод у дерева разбора *visitFrom.* Процедура передаёт методу *visitFrom* три аргумента: *VisitorMode.POST*, ссылку на метод экземпляра класса *SemanticAnalyzer* *checkExpr(n : Node<Token>)*, который удовлетворяет сигнатуре интерфейса *Action<Node<Token>>*. Метод *checkExpr* реализует проверку типов и выражений.
* *renameParams(n : LinkedNode<Token>)* – переименовывает имена параметров метода, которые встречаются в списке формальных параметров метода и в теле метода, в общие имена, используемые генератором кода.
* *setActionType(t : TranslatorAction)* – задаёт действие, которое должен выполнить данный семантический анализатор, осуществляя проход по дереву. Каждый раз, при смене действия, меняется и метод, который будет выполняться для каждой вершины дерева. Также при смене действия сбрасывается текущий контекст области видимости.
* *getAnnotatedParsedTree() : LinkedTree<Token>* - возвращает аннотированное дерево абстракного синтаксиса. Вершины такого дерева содержат экземпляры класса *TokenAttrs*, расширяющие тип *Token*, предоставляя атрибут *code*, хранящий промежуточный код. Это дерево передаётся модулю генерации промежуточного кода.
* *getCodeGenerator() : ItermediateCodeGenerator* – возвращает объект, реализующий модуль генерации промежуточного кода.
* *perform(n : Node<Token>)* – основной метод данного класса, реализация интерфейса *Action<Node<Token>*>. В нём по значению типа перечисления *TranslatorAction* определяется метод, осуществляющий определённую семантическую проверку:
  + *SEARCH\_DEFINITIONS* – осуществить проверки объявлений, вызвав процедуру *search*.
  + *RENAME\_PARAMS* – переименовать имена параметров методов везде, где они встречаются, вызвав процедуру *renameParams*.
  + *TYPE\_CHECK* – осуществить проверку типов, вызвав процедуру *typeCheck*.

Семантическая ошибка выражена типом *SemanticError*. Он имеет следующие поля, и методы:

* *msg : String* – поле, хранящее сообщение об ошибке.
* *type : SemanticErrorType* – поле, хранящее тип ошибки. Тип ошибки представлен перечислением *SemanticErrorType*, имеющее следующие значения:
  + *TYPE\_NOT\_FOUND* – используется тип, который не определён.
  + *TYPE\_REDEFINED* – объявление переменной, которая уже объявлена в текущей области видимости (переопределение).
  + *WRONG\_TYPE* – присваивание выражение с типом T1 переменной с типом T2, где тип выражения T1 не может быть приведён к типу переменной T2.
  + *WRONG\_PARAMS\_NUMBER* – число фактических параметров, и переданных аргументов метода не совпадают. Например, метод не имеет никаких параметров, но при его вызове в скобках был указан список аргументов.
  + *VARIABLE\_NOT\_FOUND* **–** используется имя переменной, которая не объявлена в текущей области видимости.
  + *VARIABLE\_NOT\_INIT –*переменная не инициализированна.
* *SemanticError(msg : String, type : SemanticErrorType)*  – конструктор, класса, создающий объект семантической ошибки, с сообщением *msg*, и типом *type*.
* *getType()* **–** возвращает тип ошибки.
* *getMsg() –*возвращает сообщение об ошибке.

Получив дерево разбора, можно осуществить семантические проверки, передав экземпляр класса семантического анализатора *SemanticAnalyzer* методу экземпляра класса дерева *LinkedTree<Token>* *visit*, указав, в каком порядке проходить узлы дерева. Но поскольку не все узлы дерева разбора содержат семантическую информацию, дерево содержит лишние вершины для семантического анализатора. Ключевые слова if, def, while, return, class являются операторами. Для того, чтобы избавиться от данных вершин, используются три класса, реализующих интерфейс *Action*. Все они меняют дерево разбора.

* *DeleteUselessSyntaxNode –* удаляет вершины из дерева, которые не содержат никакой семантики. Данный подмодуль удаляет только вершины с терминальными символами грамматики. В качестве терминалов, не содержащих семантику, могут быть символы разделители (запятая “,”, точка с запятой “;”), или группирующие круглые скобки.
* *BreakChainNode* – преобразует цепочки в дереве вида A->B->C->D в A->D. Данный класс удаляет лишние цепные вершнины, которые появляются при выполнении последовательности шагов свёрток для правил, которые обозначают приоритет операций. Цепная вершина – эта вершина, являющаяся единственным сыном родительской вершины. Кроме того, данный класс устраняет цепочки вершин, возникающих при левой или правой рекурсии продукций. Т.е. если вершина имеет родителя, содержащего грамматический символ, имя которого совпадает с именем символа, хранимым текущей вершиной, то все сыновья текущей вершины становятся сыновьями вершины родителя (а текущая вершина удаляется из дерева). Отметим, что правило для инструкций **if** и цикла **while**, содержат рекурсию, но потомки рекурсивной вершины P не добавляются к родителям, поскольку они больше отражают часть инструкций, нежели цепочку применения свёрток.
* *MakeAstTree* **–** удаляет вершины с терминальными символами, которые не являются операндом, оператором, ключевым словом, или границей области видимости. Для тех вершин, которые обозначают операторы языка, их токены копируются в родительские вершины, а сами текущие вершины удаляются из дерева.

Приведение типов осуществляет класс *TypeNegotiation*. Он определяет, содержит ли вершина численный тип, вычисляет какая из двух вершин, имеет максимальный тип, и какая из них имеет тип, больший по области значений, чем другая. Также класс снабжён методом *setGen*, позволяющий передать экземпляр класса *SemanticAnalyzer*, т.к. при приведении типов, класс *TypeNegotiation* создаёт новые временные переменные, а семантическому анализатору необходимо вести их учёт.

За генерацию кода отвечает класс *IntermediateCodeGenerator*. Он хранит в полях экземпляра ссылку на объект грамматики, аннотированного дерева разбора, счётчик меток, а также стэк имён меток, для контроля потока управления программы. Также класс хранит ссылку на экземпляр типа *FileWriter*, представляющий абстракцию потока записи символов в определённый файл. Вызвав метод *generateCode* и передав ему имя файла в виде строки, модуль генерации кода создаст текстовый файл и начнёт проходить вершины дерева в прямом порядке, и для тех вершин, которые содержат ключевое слово **def**, генерирует метку с именем метода, которое содержится в качестве второго сына узла **def**, и обходит поддерево, корнем которого является первый сын узла **def**, в обратном порядке, выполняя процедуру *produceCode* для каждого узла. Алгоритм данной процедуры представлен ниже:

1) Если текущий узел является третьим сыном узла **if**, значит было прочитано логическое выражение, от истинности которого зависит какой код будет генерироваться дальше. Необходимо сгенерировать код текущей вершины, и создать новую метку. Затем сгенерировать команду IFFALSE передав в качестве первого аргумента имя временной переменной, которая хранится в виде лексемы токена, содержащегося в текущей вершине. В качестве второго аргумента передаётся имя новой метки, а в качестве результата передаётся специальное имя “:z”, означающее отсутствие значения. Затем в сына текущей вершины, имеющий нетерминал с именем ELS, кладётся имя метки, которая сохраняется в поле *code* объекта типа *TokenAttr*.

2) Если текущий узел является вторым сыном узла **if**, значит было прочитано тело инструкции **if**.

2.1) Если данный узел не содержит токен с ключевым словом **while**, то генерируется код текущей вершины, затем в файл записывается метка из вершины ELS. В самой же вершине ELS метка удаляется.

2.2) Если данный узел содержит токен с ключевым словом **while**, значит было также обработано тело цикла **while**. Необходимо сгенерировать метку, которая хранит узел ELS. Затем необходимо переместить метку, которую хранит узел **while**, в узел ELS. Суть в том, что метка следующей команды, которая должна следовать после цикла **while**, в случае, когда выражение в условии цикла **while** является ложным, сохраняется в самом узле **while**. Это сделано, поскольку обратный порядок обхода позволяет сначала обработать выражение условия цикла **while** или инструкции **if**, затем посетить вершины, составляющих тело цикла, или инструкции **if**. Таким образом, для инструкции вида: **if** ->*B P ELS*, сначала будет посещено поддерево *B*, затем поддерево *P*, и в последнюю очередь поддерево *ELS*. Причём в каждом прогоне, имеется возможность получить родителя данных вершин, а именно: вершину **if**.

3) Если вершина является вторым сыном узла while, то значит было прочитано условие цикла. Лексема текущей вершины будет хранить имя временной переменнной для логического выражения.

**3. Технологический раздел.**

**3.1. Сборка и запуск приложения.**

Приложение собирается в единый jar файл, который затем можно запустить либо через консоль, либо двойным щелчком мыши по нажатию файла. В качестве системы сборки используется система *gradle*. Она является менее многословной и компактной, по сравнению с другими системами сборки, например такими как *maven*. Приложение написано на языке Java. Версия компилятора, входящего в набор инструментов разработчика Java (JDK) 1.8.0\_162. Код сборки, а также содержимое главного файла приложения, приведёно ниже. Для запуска сборки, необходимо в корневой папке приложения выполнить в консоли следующую команду:

*gradlew bootJar.*

Приложение сообщает пользователю о формате вводимой грамматики. Пользователь вводит абсолютное или относительное имя файла грамматики. В случае, когда он ввёл относительное имя файла, то оно будет разрешено относительно текущей рабочей директории, т.е. той папки, из которой был осуществлён запуск приложения. Приложение сообщает о результатах построения анализатора. В случае успеха, программа возвращает пользователю информацию об успешном построении и предлагает ему ввести имя входного файла для анализа. Пользователь снова вводит абсолютное или относительное имя файла. Программа выполняет синтаксический анализ, сообщая о найденных синтаксических ошибках. Если ошибок нет, то программа даёт управление модулю семантического анализа, и генератору кода. Если на семантическом анализе также обнаружены ошибки, то программа сообщает об этом пользователю и завершает свою работу. Если же ошибок нет, то программа передаёт управление дальше – модулю генерации промежуточного кода. После успешной генерации кода, пользователю вернётся текстовый файл, содержащий линейную последовательность четвёрок.

**Список литературы.**

1. БУНИНА Е.И., ГОЛУБКОВ А.Ю. Формальные языки и грамматики. Учебное пособие. – М.: Изд-во МГТУ им. Н.Э.Баумана, Москва, 2006. URL: <http://iu9.bmstu.ru/data/book/fl.pdf>

2. ECMA-404, The JSON Data Interchange Syntax 2nd edition.

3. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т.1.:Синтаксический анализ. –М.: Мир,1978.

4. Right recursion versus left recursion // IBM Knowledge Center [Электронный ресурс]. URL: <https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.bpxa600/rrlr.htm>

5. Eliminating left-recursion: three steps. URL: http://www.d.umn.edu/~hudson/5641/l11m.pdf

6. Ахо А.В., Лам М.С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструменты. –М.: Вильямс, 2008.

7. DFA minimization.

URL: <https://en.wikipedia.org/wiki/DFA_minimization#Hopcroft's_algorithm>

7. D. Grune, C. H. J. Jacobs "Parsing Techniques - A Practical Guide", Ellis Horwood, 1990.

URL: http://www.cs.vu.nl/~dick/PTAPG.html

8. Introduction to Recursive Descent Parsing.

URL: http://ag-kastens.uni-paderborn.de/lehre/material/uebi/parsdemo/recintro.html

9. Parsing Expressions by Recursive Descent.

URL: http://www.engr.mun.ca/~theo/Misc/exp\_parsing.htm

10. Молчанов А. Ю. Системное программное обеспечение. Учебник для вузов. 3-е изд. –Спб.: Питер, 2010. -400с.: ил.

11. Молчанов А. Ю. Системное программное обеспечение. Лабораторный практикум. – Спб.: Питер, 2005. - 284с.: ил.

**Приложения.**

**Приложение 1.1. Код модуля разбора JSON-файлов.**