

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра «Математическое обеспечение и применение ЭВМ»

Курсовая работа
по дисциплине «Теория языков программирования и методы трансляции»
на тему «Разработка транслятора»

ПГУ 09.03.04 - 05КР211.05 ПЗ

Направление подготовки – 09.03.04 Программная инженерия

Выполнил студент:
Группа:



Шахманов Д.Д.
21ВП1

Руководитель:
к.т.н., доцент



Дорофеева О.С.

Работа защищена с оценкой

отлично

Преподаватели



Дата защиты

8.12.2023

Пенза, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ЗАДАНИЕ

на курсовой проект по дисциплине

«Теория языков программирования и методы трансляции»

Тема: «Разработка транслятора»

ВАРИАНТ 5

На основании базового описания языка и в соответствии с вариантом задания разработать транслятор с заданного языка. Базовое описание языка имеет следующий вид:

<Программа> ::= <Объявление переменных> <Описание вычислений>
<Описание вычислений> ::= BEGIN <список присваиваний> END
<Объявление переменных> ::= VAR <список переменных> : тип ;
<Список переменных> ::= <Идент>|<Идент>,<Список переменных>
<Список присваиваний> ::= <Присваивание>|<Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> = <Выражение>;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= (<Выражение>) | <Операнд> | <Подвыражение>
<Бин.оп.><Подвыражение>
<Ун.оп> ::= вид
<Бин.оп.> ::= вид
<Операнд> ::= <Идент>|<Конст>
<Идент> ::= <Буква><Идент>|<Буква>
<Конст> ::= вид

Необходимо использовать следующие данные:

Тип переменных	Вид <Ун.оп.>	Вид <Бин.оп.>	Вид <конст.>	Макс. длина идентификатора	Распознаватель
INTEGER	-	+ - *	<цифра> <конст> <цифра>	10	Нисходящий

Необходимо использовать следующие операторы:

- 1) READ(<Список переменных>);
- 2) FOR <Идент>=<Выражение> TO <Выражение> DO
<Список присваиваний> END_FOR;
- 3) WRITE(<Список переменных>).

Разработка будет производиться на языке программирования C#, в среде Visual Studio 2022.

Руководитель работы, к.т.н., доцент кафедры



Дорофеева О. С.

г. 09. 2023г.

Реферат

Пояснительная записка содержит лист, рисунков, таблицы, использованных источника, приложение.

ТРАНСЛЯТОР, ИНТЕРПРЕТАТОР, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, НИСХОДЯЩИЙ РАЗБОР, ГРАММАТИКА LL(1), ФУНКЦИИ ПЕРЕХОДОВ.

Объектом исследования являются формальные грамматики.

Целью курсовой работы является разработка транслятора типа интерпретатор с языка, определённого соответствующей формальной грамматикой.

Разработка проводилась на языке программирования C# в среде программирования Visual Studio 2022.

Осуществлено функциональное тестирование разработанного программного обеспечения, которое показало корректность его работы.

					ПГУ 09.03.04 - 05КР201.05 ПЗ						
Изм.	Лист	№ докум.	Подп	Дата							
Разраб.		Шахманов Д. Д.			«Разработка транслятора» Пояснительная записка.			Лит.	Лист	Лист	
Пров.		Дорофеева О.С.								3	46
Т. контр.								Группа 21ВП1			
Н. контр.											
Утв.											

Содержание

Введение.....	4
1. Методы грамматического разбора	6
1.2 Разбор сверху-вниз.....	7
1.2.1 LL(k) — языки и грамматики	7
1.2.2 Метод рекурсивного спуска.....	8
2 Разработка модулей транслятора	11
2.1 Анализ требований.....	11
2.1.1 Требования к интерфейсу пользователя.....	11
2.1.2 Требования к программным средствам	11
2.2 Проектирование	15
2.2.1 Проектирование лексического анализатора	15
2.2.2 Приведение грамматики к виду LL(1)	17
2.2.3 Программная реализация синтаксического анализатора.....	18
2.2.4 Программная реализация компилятора	21
2.2 Кодирование	23
2.4 Тестирование.....	25
2.4.1 Виды тестирования	25
2.4.2 Функциональное тестирование	25
Заключение	33
Список использованных источников.....	34
Приложение А. Листинг программного текста транслятора.....	35
Приложение Б. Диаграмма классов	Error! Bookmark not defined.

Введение

Современные информационные технологии напрямую зависят от языков программирования, поскольку всё программное обеспечение на всех компьютерах написано на том или ином языке программирования. Языки программирования представляют собой средство описания вычислений и алгоритмов в виде, удобном как для чтения человеком, так и для представления в ЭВМ.

В настоящее время существует множество разнообразных языков программирования. Их практическое использование невозможно без соответствующей системы программирования, основу которой составляет транслятор. Чтобы программа могла быть выполнена на ЭВМ, ее необходимо преобразовать в форму, которая может это обеспечить.

Как известно, целью трансляции является преобразование исходного текста программы в текст, который будет понятен адресату. В качестве адресата может выступать как программное, так и техническое средство. Следовательно, с развитием вычислительных систем разработка качественного транслятора остаётся актуальной темой. Известно, что транслятор имеет ряд характеристик:

- корректная обработка исходного(входного) текста;
- корректная обработка всевозможных исключительных ситуаций;
- универсальность;
- оптимизированная работа;
- наличие на выходе корректного результата обработки исходного текста.

Выше перечисленные пункты значимы при разработке, потому что транслятор, который будет некорректно обрабатывать входные данные, или иметь на выходе ложный результат, никому не нужен. Так же очень важна скорость обработки входных данных, поэтому оптимизация играет не малую роль.

Целью данной курсовой работы является разработка транслятора.

Исходя из цели данной курсовой работы, были поставлены следующие задачи:

- построить формальную грамматику, привести её к виду LL(1);
- разработать лексический анализатор, на выходе которого формируется последовательность лексем;
- разработать синтаксический анализатор для нисходящего грамматического разбора;
- разработать компилятор;
- провести тестирование программного кода и выявить все возможные виды поведения пользователя;
- проанализировать результаты и сделать выводы о работе программного обеспечения.

Для разработки данной курсовой работы был выбран язык программирования C#. Для разработки программного кода и пользовательского интерфейса была выбрана среда разработки Visual Studio 2022

1. Методы грамматического разбора

Во время синтаксического анализа предложения исходной программы распознаются как языковые конструкции, описываемые используемой грамматикой. Этот процесс можно рассматривать как построение дерева грамматического разбора для транслируемых предложений.

Методы грамматического разбора разбиваются на два больших класса - восходящие и нисходящие - в соответствии с порядком построения дерева грамматического разбора.

Нисходящие методы (методы сверху вниз) начинают с правила грамматики, определяющего конечную цель анализа с корня дерева грамматического разбора, и пытаются так его наращивать, чтобы последующие узлы дерева соответствовали синтаксису анализируемого предложения.

Восходящие методы (методы снизу-вверх) начинают с конечных узлов дерева грамматического разбора и пытаются объединить их построением узлов все более и более высокого уровня до тех пор, пока не будет достигнут корень дерева. [1]

1.2 Разбор сверху-вниз

Нисходящий метод называют рекурсивным спуском. Это один из методов грамматического анализа, где правила формальной грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности токенов.

Процессор грамматического разбора, основанный на этом методе, состоит из отдельных процедур для каждого нетерминального символа, определенного в грамматике. Каждая такая процедура носит имя нетерминала и ищет во входном потоке подстроку, начинающуюся с текущей лексемы, которая может быть интерпретирована как нетерминальный символ, связанный с данной процедурой. В процессе своей работы она может вызывать другие подобные процедуры или даже рекурсивно саму себя для поиска других нетерминальных символов.

Если эта процедура находит соответствующий нетерминальный символ, то она заканчивает свою работу, передает в вызвавшую ее программу признак успешного завершения и устанавливает указатель текущей лексемы на первую лексему после распознанной подстроки.

Если же процедура не может найти подстроку, которая могла бы быть интерпретирована как требуемый нетерминальный символ, она заканчивается с признаком ошибки или вызывает процедуру выдачи диагностического сообщения и процедуру восстановления.

Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена. [2]

1.2.1 LL(k) — языки и грамматики

Моделирование работы недетерминированных магазинных распознавателей связано с поиском последовательности переходов из

начального в одно из конечных состояний. Поиск состоит из отдельных шагов, каждый из которых может закончиться неудачно и привести к возврату в исходное состояние для выполнения следующего шага. Такой поиск с возвратом связан со значительными затратами времени, поэтому на практике используют более экономичные детерминированные распознаватели, работающие без возвратов. Эти распознаватели допускают только ограниченные классы КС языков, которые, однако, отражают все синтаксические черты языков программирования.

В общем случае грамматика относится к классу $LL(K)$ грамматик, если для нее можно построить нисходящий детерминированный распознаватель, учитывающий K входных символов, расположенных справа от текущей входной позиции.

Название LL произошло от слова *Left*, поскольку анализатор просматривает входную цепочку слева направо, и слова *Leftmost*, поскольку он обнаруживает появление правила по одному или группе символов, образующих левый край цепочки. На практике наибольшее применение имеет класс $LL(1)$ грамматик, для которых детерминированный распознаватель работает по одному входному символу, расположенному в текущей позиции. [3]

1.2.2 Метод рекурсивного спуска

Метод рекурсивного спуска (РС-метод) - для каждого нетерминала грамматики создается своя процедура, носящая его имя; ее задача – начиная с указанного места исходной строки найти подстроку, которая выводится из этого нетерминала. Если такую подстроку считать не удастся, то процедура завершает свою работу вызовом процедуры обработки ошибки, которая выдает сообщение о том, что строка не принадлежит языку, и останавливает разбор. Если подстроку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры пишется непосредственно по правилам вывода

соответствующего нетерминала: для правой части каждой продукции осуществляется поиск подстроки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

Метод рекурсивного спуска применим в том случае, если каждое правило грамматики имеет вид:

- а) $A \rightarrow \alpha$, где $\alpha (T N)^*$, и это единственное правило вывода для этого нетерминала;
- б) $A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \dots \mid a_n\alpha_n$, где $a_i \in T$ для всех $i = 1, 2, \dots, n$; $a_i \neq a_j$ для $i \neq j$; $\alpha_i (T N)^*$, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Ясно, что если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по вышеизложенной схеме. Если грамматика не удовлетворяет этим условиям, то это алгоритмически неразрешимая проблема, т.е. эквивалентной КС-грамматики, для которой метод рекурсивного спуска применим, нет.

Изложенные выше ограничения являются достаточными, но не необходимыми. Ослабление требований на вид правил грамматики:

1. При описании синтаксиса языков программирования часто встречаются правила, описывающие последовательность однотипных конструкций, отделенных друг от друга каким-либо знаком-разделителем (например, список идентификаторов при описании переменных, список параметров при вызове процедур и функций и т.п.).

Общий вид этих правил: $L \rightarrow a \mid a, L$.

Формально здесь не выполняются условия применимости метода рекурсивного спуска, так как две альтернативы начинаются одинаковыми терминальными символами.

Действительно, в строке a, a, a, a из нетерминала L может выводиться и подстрока a , и подстрока a, a , и вся строка a, a, a, a . Неясно, какую из них

выбрать в качестве подстроки, выводимой из L . Если принять решение, что в таких случаях будем выбирать самую длинную подстроку (что и требуется при разборе реальных языков), то разбор становится детерминированным. [4]

2 Разработка модулей транслятора

2.1 Анализ требований

2.1.1 Требования к интерфейсу пользователя

Разработанное приложение должно предоставлять обработку программы, записанной пользователем в файл с расширением txt, а также должно предоставлять консольный интерфейс, позволяющий:

1. Выводить результаты работы программы, описанной пользователем в терминал.
2. Выводить ошибки описания программы на различных этапах анализа или трансляции программы: на этапе лексического анализа, или на этапе синтаксического анализа, или на этапе трансляции.

2.1.2 Требования к программным средствам

Разработанное приложение должно позволять:

- принимать исходный текст программы в формате txt;
- осуществлять лексический анализ исходного текста программы и составить из него последовательность лексем;
- осуществлять синтаксический анализ последовательности лексем и составить из нее специальную форму записи программы;
- последовательно интерпретировать действия из промежуточной записи программы и выполнять их;
- выводить результат работы программы в стандартный поток вывода.

Диаграмма вариантов использования, удовлетворяющая данным требованиям, приведена на рисунке 1.

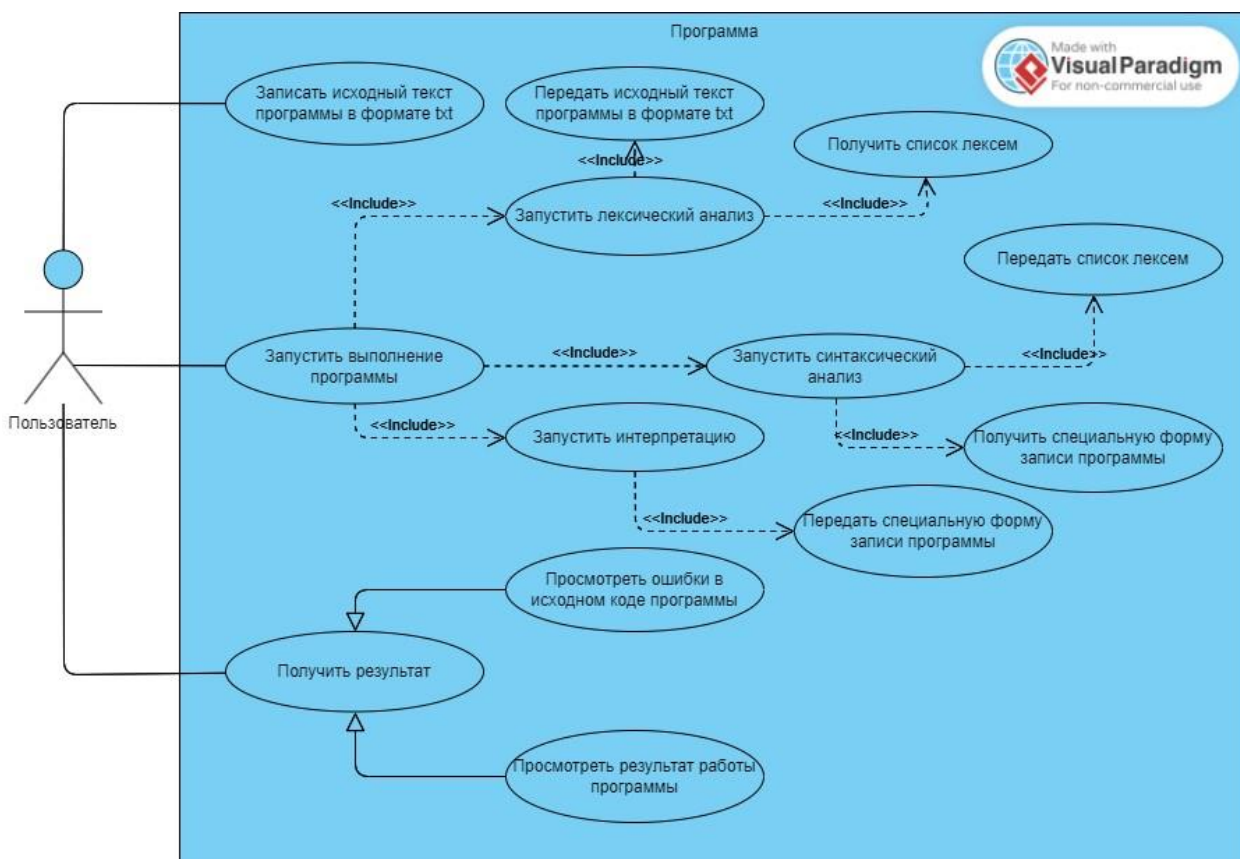


Рисунок 1 – Диаграмма вариантов использования

В результате анализа поставленной задачи были выделены варианты использования: «Записать исходный текст программы в формате txt»; «Запустить выполнение программы», который включает в себя такие варианты использования, как: «Запустить лексический анализ», «Запустить синтаксический анализ», «Запустить интерпретацию»; «Получить результат», который обобщает варианты использования «Просмотреть ошибки в исходном коде программы» и «Просмотреть результат работы программы». В свою очередь вариант использования «Запустить лексический анализ» включает в себя «Передать исходный текст программы в формате txt» и «Получить список лексем», вариант использования «Запустить синтаксический анализ» включает в себя вариант использования «Получить специальную форму записи программы» и «Передать список лексем», а «Запустить интерпретации» включает в себя «Передать специальную форму записи программы».

Сценарии вариантов использования приведены в таблицах 1 – 3.

Таблица 1 – Спецификация прецедента «Запустить лексический анализ»

Наименование: Провести лексический анализ
ID: 1
Краткое описание: на вход лексического анализатора поступает исходный текст программы в формате txt, который подвергается лексическому анализу
Действующие лица: программа, пользователь
Предусловие: пользователь сформировал исходный текст программы в формате txt и передал его программе
Основной поток: <ol style="list-style-type: none"> 1. программе на вход поступает исходный текст программы в формате txt; 2. программа из исходного текста программы лексемы последовательно извлекаются, распознаются по принадлежности к определённому типу (переменные, операции и т. д.) и записываются в список лексем; 3. программа возвращает упорядоченный список лексем или генерирует исключение при ошибке.
Постусловие: получен упорядоченный список лексем или сгенерировано исключение при ошибке лексического анализа

Таблица 2 – Спецификация прецедента «Запустить синтаксический анализ»

Наименование: Провести синтаксический анализ
ID: 2
Краткое описание: программа проводит синтаксический анализ упорядоченного списка лексем из списка лексем, а также создаёт промежуточную форму записи программы

Продолжение таблицы 2

Действующие лица: программа, пользователь
Предусловие: Пользователь сформировал упорядоченный список лексем
Основной поток: 1. программа, используя функции перехода, анализирует список лексем; 2. при удачном анализе создаёт специальную форму записи программы или генерирует исключение при ошибке
Постусловие: была получена специальная форма записи программы или сгенерировано исключение при ошибке синтаксического анализа

Таблица 3 – Спецификация прецедента «Запустить интерпретацию»

Наименование: Распознать псевдокод
ID: 3
Краткое описание: Программа покомандно выполняет действия из специальной формы записи программы
Действующие лица: программа, пользователь
Предусловие: пользователь сформировал специальную форму записи программы
Основной поток: 1. пользователь передает специальную форму записи программы; 2. программа покомандно выполняет действия из специальной формы записи программы; 3. программа успешно заканчивает выполнение специальной формы записи программы или генерирует исключение при ошибке.
Постусловие: получен список токенов исходная программа, полученная от пользователя выполнена или сгенерировано исключение при ошибке в ходе трансляции программы.

2.2 Проектирование

2.2.1 Проектирование лексического анализатора

Лексический анализ включает в себя сканирование исходного текста программы путем поиска определенных токенов, которые будут помещены в упорядоченный список, представляющий собой упорядоченный список лексем.

Лексема - минимальная единица смысла в языке программирования или естественном языке. Она представляет собой отдельное слово или символьную последовательность, которая имеет определенное значение. Лексема не зависит от контекста и может быть любым значимым элементом, таким как ключевые слова, идентификаторы, числа, операторы и т.д.

Токен - единица анализа текста, которая представляет собой лексему или символ, используемый в процессе обработки текста. Это более абстрактная единица, которая может включать в себя не только лексемы, но и другие символы, такие как пробелы и знаки препинания.

В программе сущность токен представлена классом `Token` и расширяющим классом `ValuedToken`.

```
public class Token {
    public TokenType type;
}

public class ValuedToken : Token {
    public string Value;
}
```

Класс `Token` используется для лексем, представляющих собой терминальный символ, то есть символ, который не может быть заменен на другие символы в рамках грамматики. Он представляет конкретные элементы языка. (например лексема объявления переменных «VAR», знак сложения «+»).

Класс ValuedToken используется для лексем, представляющих собой нетерминальные символ, то есть символ, который может быть заменен на последовательность других терминальных или нетерминальных символов в рамках грамматики (например лексема число может быть представлена последовательностью символов [0-9]+)

Для заданного описания языка определены типы токенов (TokenType) и соответствующие им лексемы (таблица 4). Токен типа ignore представляет собой символ/последовательность символов пробел и игнорируется при лексическом анализе.

Таблица 4 – Соответствия токен - лексема

TokenType	Лексема	
	Значение	Назначение
var	var VAR	Keyword
integer	integer INTEGER	Keyword
begin	begin BEGIN	Keyword
end	end END	Keyword
to	to TO	Keyword
for	for FOR	Keyword
do	do DO	Keyword
read	read READ	Keyword
write	write WRITE	Keyword
end_for	end_for END_FOR	Keyword
semicolon	;	Separator
comma	,	Separator
identifier	[a-z]{1,12}	Identifier
number	d+	Number
colon	:	Special symbol
close_parenthesis)	Special symbol
open_parenthesis	(Special symbol
addition_operator	+	Operator
subtraction_operator	-	Operator
multiplication_operator	*	Operator
division_operator	/	Operator
negate_operator	!	Operator
assignment_operator	=	Operator
ignore		

Из исходной программы токены последовательно извлекаются, распознаются по принадлежности к определённому типу (переменные, операции и т. д.) и записываются в список лексем (кроме токена типа ignore).

Результатом работы лексического анализатора является список лексем или вывод лексических ошибок в исходном тексте программы (использование недопустимого символа, превышение длины идентификатора).

2.2.2 Приведение грамматики к виду LL(1)

Для нисходящего грамматического разбора для грамматики типа LL(1) потребовалось устранить прямую левую рекурсию и провести факторизацию.

В таблице 5 представлен словарь нетерминальных символов.

Таблица 5 – Словарь нетерминальных символов

Нетерминал	Обозначение
<Программа>	Program
<Список операций>	Statements
<Список переменных>	VariableList
<Идент>	Identifier
<Операция>	Statement
<Операция присваивания>	AssignmentStatement
<Начало выражений>	Expression
<Выражение>	Expression`
<Начало членов выражения>	Term
<Член выражения>	Term`
<Множитель выражения>	Factor
<Числовой литерал>	NumberLiteral
<Операция вывода в консоль>	WriteStatement
<Операция считывания с консоли>	ReadStatement
<Операция цикла for>	ForStatement

Проведя требуемые преобразования получили правила грамматики LL(1), которые представлены в таблице 6.

Таблица 6 – Грамматика языка

Program	::= "VAR" VariablesList ":" "INTEGER" ";" "BEGIN" Statements "END"
VariablesList	::= Identifier ("," Identifier)*
Identifier	::= [a-z]{1,12}
Statements	::= Statement+
Statement	::= AssignmentStatement ReadStatement WriteStatement ForStatement
AssignmentStatement	::= Identifier "=" Expression ";"
Expression	::= Term SubExpression
Expression`	::= "+" Term SubExpression "-" Term SubExpression ϵ
Term	::= Factor SubTerm
Term`	::= "*" Factor SubTerm "/" Factor SubTerm ϵ
Factor	::= "!" Factor {1} NumberLiteral Identifier "(" Expression ")"
NumberLiteral	::= /d+
ForStatement	::= "FOR" Identifier "=" Expression "TO" Expression "DO" Statements "END_FOR"
WriteStatement	::= "WRITE" "(" VariablesList ")" ";"
ReadStatement	::= "READ" "(" VariablesList ")" ";"

2.2.3 Программная реализация синтаксического анализатора

Синтаксический анализатор считывает список лексем, формируемый лексическим анализатором, осуществляет грамматический разбор, выдает сообщения о синтаксических ошибках при их наличии и создает промежуточную форму записи исходной программы. Основой разработки синтаксического анализатора является проектирование и реализация соответствующего магазинного автомата.

Для нисходящего грамматического разбора, для грамматики типа LL(1) после приведения ее к нужному виду требуется спроектировать магазинный автомат с подробным описанием всех переходов в рамках функции переходов.

Функция перехода магазинного автомата имеет вид:

$(State, Input, StackTop) \Rightarrow (NewState, StackOperation)$

1. State - текущее состояние магазинного автомата;
2. Input - символ входной последовательности;
3. StackTop - символ в вершине магазина;
4. NewState - состояние магазинного автомата после перехода;
5. StackOperation - операции с магазином

Операции с магазином магазинного автомата:

1. push(value) - добавляет символ value в вершину магазина магазинного автомата.
2. pop(n) - удаляет n символов с вершины магазина магазинного автомата.
3. clear() - очищает магазин магазинного автомата.

В процессе реализации синтаксического анализатора было выявлено 38 функций перехода и 10 состояний.

Алфавит магазинного автомата был дополнен символами:

1. MainStatements - символ указывающий на то, что операции выполняются внутри тела программы;
2. ForStatements - символ указывающий на то, что операции выполняются внутри тела цикла for;
3. IterAssignment - символ указывающий на то, что операция присвоения происходит для итератора цикла for.

Состояния магазинного автомата:

1. S1 - начальное состояние, состояние считывания программы;
2. S2 - состояние считывания объявления переменных;
3. S3 - состояние выбора одной из операций;
4. S4 - состояние считывания операции считывания с консоли;
5. S5 - состояние считывания операции вывода в консоль;
6. S6 - состояние считывания операции присвоения;
7. S7 - состояние считывания операции цикл for;

8. S8 - состояние выбора одной из операций внутри цикла for;
9. S9 - состояние выбора возвращения к телу программы или к телу цикла for;
10. S10 - состояние успешного завершения считывания программы.

Функции переходов:

* – пустой такт, входной символ ленты не считывается / операции с магазином магазинного автомата отсутствуют.

1. (S1, "VAR", h0) => (S2, push("VAR"))
2. (S2, VariableList, "VAR") => (S2, push(VariableList))
3. (S2, ":", VariableList) => (S2, push(":"))
4. (S2, "INTEGER", ":") => (S2, push("INTEGER"))
5. (S2, ";;", "INTEGER") => (S1, pop(4); push(Declaration))
6. (S1, "BEGIN", Declaration) => (S3, MainStatements)
7. (S3, "READ", MainStatements) => (S4, "READ")
8. (S4, "(", "READ") => (S4, "(")
9. (S4, VariableList, "(") => (S4, VariableList)
10. (S4, ")", VariableList) => (S4, ")")
11. (S4, ";;", ")") => (S3, pop(4))
12. (S3, "WRITE", MainStatements) => (S5, "WRITE")
13. (S5, "(", "WRITE") => (S4, "(")
14. (S5, VariableList, "(") => (S5, VariableList)
15. (S5, ")", VariableList) => (S5, ")")
16. (S5, ";;", ")") => (S3, pop(4))
17. (S3, Identifier, MainStatements) => (S6, Identifier)
18. (S6, "=", Identifier) => (S6, "=")
19. (S6, Expression, "=") => (S6, Expression)
20. (S6, ";;", Expression) => (S3, pop(4))
21. (S6, "TO", Expression) => (S7, pop(4))
22. (S3, "FOR", MainStatements) => (S7, push("FOR"))

23. (S7, Identifier, "FOR") => (S8, push(IterAssignment); push(Identifier))
24. (S7, "TO", IterAssignment) => (S7, push("TO")))
25. (S7, Expression, "TO") => (S7, push(Expression))
26. (S7, "DO", Expression) => (S8, push(ForStatements))
27. (S7, *, "END_FOR") => (S8, pop(1))
28. (S8, "READ", ForStatements) => (S4, push("READ")))
29. (S8, "WRITE", ForStatements) => (S5, push("WRITE")))
30. (S8, Identifier, ForStatements) => (S6, push(Identifier))
31. (S8, "FOR", ForStatements) => (S7, push("FOR")))
32. //pop symbols ["FOR", ...)
33. (S8, "END_FOR", ForStatements) => (S8, pop(5); push("END_FOR")))
34. (S8, *, "END_FOR") => (S9, pop(1))
35. (S9, *, ForStatements) => (S7, push("END_FOR")))
36. (S9, *, MainStatements) => (S3, *)
37. (S3, "END", MainStatements) => (S1, *)
38. (S1, *, MainStatements) => (S10, clear())

2.2.4 Программная реализация интерпретатора

Интерпретатор анализирует и тут же выполняет программу покомандно или построчно по мере поступления её исходного кода на вход интерпретатора.

Алгоритм работы интерпретатора:

1. прочитать инструкцию;
2. проанализировать инструкцию и определить соответствующие действия;
3. выполнить соответствующие действия;
4. если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.

Достоинства использования интерпретатора

Большая переносимость интерпретируемых программ — программа будет работать на любой платформе, на которой реализован соответствующий интерпретатор.

Как правило, более совершенные и наглядные средства диагностики ошибок в исходных кодах.

Меньшие размеры кода по сравнению с машинным кодом, полученным после обычных компиляторов.

Недостатки использования интерпретатора

Интерпретируемая программа не может выполняться отдельно без программы-интерпретатора. Сам интерпретатор при этом может быть очень компактным.

Интерпретируемая программа выполняется медленнее, поскольку промежуточный анализ исходного кода и планирование его выполнения требуют дополнительного времени в сравнении с непосредственным исполнением машинного кода, в который мог бы быть скомпилирован исходный код.

В данной курсовой работе интерпретатор покомандно выполняет действия из специальной формы записи программы, полученной из синтаксического анализатора, интерпретируя их в форму записи языка C#.

2.2 Кодирование

Структура программного обеспечения описана на диаграмме (Приложение Б).

В ходе выполнения курсовой работы было разработано приложение, код данного приложения приведен в приложении А.

Класс **Lexer** выполняет лексический анализ.

Класс **Parser** выполняет синтаксический анализ.

Класс **Interprete** осуществляет трансляцию программы.

Было выявлено 4 основных операции для выполнения в теле программы: операция присвоения значения переменной (**AssignmentStatement**), операция считывания ввода с консоли (**ReadStatement**), операция вывода информации в консоль (**WriteStatement**) и операция выполнения цикла for (**ForStatement**).

Эти операции реализуют **интерфейс IStatement**:

```
internal interface IStatement {
    public StatementType Type;
}
```

Type - тип операции.

Операция присвоения значения переменной:

```
internal class AssignmentStatement : IStatement {
    public StatementType Type;
    public ValuedToken Variable;
    public List<Token> ValueInfixExpression;
}
```

Variable - переменная, которой будет присвоено значение (лексема с типом IDENTIFIER)

ValueInfixExpression - выражение в инфиксной форме для вычисления значения (упорядоченный список лексем, представляющий собой алгебраическое выражение в инфиксной форме, содержит лексемы с типами NUMBER, IDENTIFIER, ADDITION_OPERATOR и т.д.).

Операция вывода информации в консоль:


```
internal class WriteStatement : IStatement {
    public StatementType Type;
    public List<ValuedToken> Variables;
}
```

Variables - переменные, значения которых будут выведены в консоль (список лексем с типом IDENTIFIER).

Операция считывания ввода с консоли:

```
internal class ReadStatement : IStatement {
    public StatementType Type;
    public List<ValuedToken> Variables;
}
```

Variables - переменные, значения которых будут введены с консоли (список лексем с типом IDENTIFIER).

Операция выполнения цикла for:

```
internal class ForStatement : IStatement {
    public StatementType Type;
    public AssignmentStatement IteratorStatement;
    public List<Token> EndIteratorValueExpression;
    public List<IStatement> Statements;
}
```

IteratorStatement - операция присвоения начального значения переменной, использующейся как итератор.

EndIteratorValueExpression - выражение в инфиксной форме для вычисления конечного значения итератора (упорядоченный список лексем, представляющий собой алгебраическое выражение в инфиксной форме, содержит лексемы с типами NUMBER, IDENTIFIER, ADDITION_OPERATOR и т.д.).

Statements - список операций, которые содержит тело цикла for.

2.4 Тестирование

2.4.1 Виды тестирования

Тестирование – это процесс исследования (испытания) программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов. Существуют различные типы тестирования:

1. Модульные тесты

Модульные тесты работают на очень низком уровне, близко к исходному коду приложения. Они заключаются в тестировании отдельных методов и функций классов, компонентов или модулей, используемых в ПО. Модульные тесты, как правило, не требуют больших расходов на автоматизацию и могут выполняться сервером непрерывной интеграции очень быстро. Интеграционное тестирование В ходе интеграционного тестирования проверяется, хорошо ли работают вместе различные модули и сервисы, используемые приложением. Например, можно протестировать взаимодействие с базой данных или убедиться, что микросервисы работают вместе так, как задумано. Этот вид тестирования является более затратным, поскольку для проведения тестов требуется запуск различных компонентов приложения.

2. Функциональные тесты

В функциональных тестах основное внимание уделяется бизнес-требованиям к приложению. Они проверяют только результат некоторого действия и не проверяют промежуточные состояния системы при выполнении этого действия. Иногда возникает путаница между понятиями интеграционных и функциональных тестов, так как и те и другие требуют взаимодействия нескольких компонентов друг с другом. Разница в том, что интеграционный тест нужен просто чтобы убедиться, что вы можете отправлять запросы к базе данных, тогда как функциональный тест будет

ожидать получения из базы данных определенного значения в соответствии с требованиями продукта.

3. Сквозные тесты

Сквозное тестирование копирует поведение пользователя при работе с ПО в контексте всего приложения. Оно обеспечивает контроль того, что различные схемы действий пользователя работают должным образом. Сценарии могут быть как очень простыми (загрузка веб-страницы или вход в систему), так и гораздо более сложными (проверка почтовых уведомлений, онлайн-платежей и т. д.). Сквозные тесты очень полезны, но их выполнение обходится довольно дорого, к тому же, когда они автоматизированы, такие тесты тяжело обслуживать. Рекомендуется иметь в наличии несколько основных сквозных тестов и активнее полагаться на более низкие уровни тестирования (модульные и интеграционные тесты), чтобы получать возможность быстро выявлять критические изменения.

4. Приемочное тестирование

Приемочные тесты — это формальные тесты, которые проверяют, отвечает ли система требованиям бизнеса. При этом во время тестирования должно быть запущено само приложение, и основное внимание уделяется воспроизведению поведения пользователей. В ходе этого тестирования возможен даже замер производительности системы, и в случае несоответствия установленным требованиям внесенные изменения могут быть отклонены.

5. Тестирование производительности

В тестах производительности оценивается работа системы при определенной рабочей нагрузке. С помощью таких тестов можно оценить надежность, скорость, масштабируемость и отзывчивость приложения. Например, это может быть наблюдение за временем отклика при выполнении большого количества запросов или определение поведения системы при работе со значительными объемами данных. Этот вид тестирования позволяет определить, соответствует ли приложение требованиям к

производительности, найти узкие места, оценить стабильность при пиковом трафике и многое другое.

6. Smoke-тестирование

Smoke-тесты — это базовые тесты, которые проверяют основные функциональные возможности приложения. Они должны выполняться быстро, поскольку цель таких тестов — убедиться, что основные возможности системы работают как запланировано. Smoke-тесты полезно запускать сразу после создания новой сборки (для определения, можно ли запускать более ресурсоемкие тесты) или сразу после развертывания (чтобы убедиться, что приложение работает правильно в новой, только что развернутой среде).

2.4.2 Функциональное тестирование

Для использования в данной курсовой работе был выбран метод функционального тестирования.

Тест №1.

Состав теста: некорректное имя переменной (длина имени >12 символов)

Ожидаемый результат: вывод сообщения о некорректном имени переменной.

Входной текст программы:

```
var n, a, b, c, d, f, aaaaabbbbbccd : integer;
begin
    a=5;
end
```

Результат: в консоль выведено сообщение о некорректном имени переменной (Рисунок 2).

```
Not correct identifier: type: IDENTIFIER, value: aaaaabbbbbccd, max length of
identifier is 12 symbols
```

Рисунок 2 – Некорректное имя переменной

Тест №2.

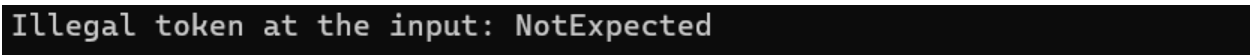
Состав теста: наличие недопустимого символа в исходном тексте.

Ожидаемый результат: вывод сообщения о недопустимом символе.

Входной текст программы:

```
var n, a, b, c, d, f : integer;
    begin
        NotExpected
        a=5;
    end
```

Результат: в консоль выведено сообщение о недопустимом символе (Рисунок 3).



```
Illegal token at the input: NotExpected
```

Рисунок 3 – Недопустимый символ

Тест №3.

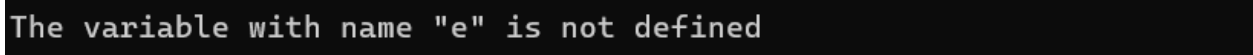
Состав теста: использование необъявленной переменной.

Ожидаемый результат: вывод сообщения о необъявленной переменной.

Входной текст программы:

```
var n, a, b, c, d, f : integer;
    begin
        e=5;
    end
```

Результат: в консоль выведено сообщение о необъявленной переменной. (Рисунок 4).



```
The variable with name "e" is not defined
```

Рисунок 4 – Необъявленная переменная

Тест №4.

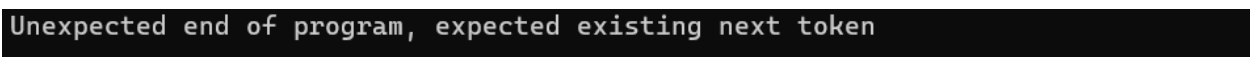
Состав теста: некорректное окончание программы. Отсутствие end.

Ожидаемый результат: вывод сообщения о некорректном окончании программы.

Входной текст программы:

```
var n, a, b, c, d, f : integer;
begin
    a=5;
```

Результат: в консоль выведено сообщение о некорректном окончании программы. (Рисунок 5).



```
Unexpected end of program, expected existing next token
```

Рисунок 5 – Некорректное окончание программы

Тест №5.

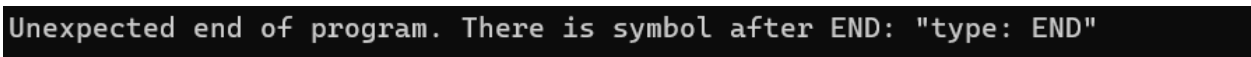
Состав теста: наличие лексем после end.

Ожидаемый результат: вывод сообщения о наличии лексем после end.

Входной текст программы:

```
var n, a, b, c, d, f : integer;
begin
    a=5;
end
end
```

Результат: в консоль выведено сообщение о наличии лексем после end. (Рисунок 6).



```
Unexpected end of program. There is symbol after END: "type: END"
```

Рисунок 6 – Наличие лексем после end

Тест №6.

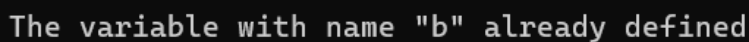
Состав теста: Повторное объявление переменной.

Ожидаемый результат: вывод сообщения о повторном объявлении переменной.

Входной текст программы:

```
var n, a, b,b, c, d, f : integer;
begin
    a=5;
end
```

Результат: в консоль выведено сообщение о повторном объявлении переменной. (Рисунок 7).



The variable with name "b" already defined

Рисунок 7 – Повторное объявление переменной

Тест №7.

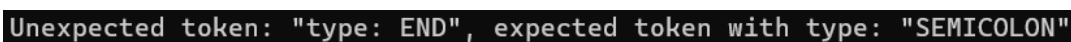
Состав теста: Пропуск разделительного символа «;».

Ожидаемый результат: вывод о пропуске разделительного символа «;».

Входной текст программы:

```
var n, a, b,b, c, d, f : integer;
begin
    a=5;
    write(a)
end
```

Результат: в консоль выведено сообщение о пропуске разделительного символа «;». (Рисунок 8).



Unexpected token: "type: END", expected token with type: "SEMICOLON"

Рисунок 8 – Пропуск разделительного символа

Тест №.8

Состав теста: Проверка операции цикл for.

Ожидаемый результат: корректное выполнение операций, описанных внутри тела цикла for заданное количество раз.

Входной текст программы:

```
var n, a, b, c, d, i, j : integer;
begin
    a=5;
    for i=0 to 3 do
        for j=0 to 3 do
            c=j+i;
            write(c);
        end_for
    end_for
end
```

Результат: операции, описанные внутри цикла for выполнились корректно заданное количество раз. (Рисунок 9).

```
Value of variable with name "c" is 0
Value of variable with name "c" is 1
Value of variable with name "c" is 2
Value of variable with name "c" is 1
Value of variable with name "c" is 2
Value of variable with name "c" is 3
Value of variable with name "c" is 2
Value of variable with name "c" is 3
Value of variable with name "c" is 4
```

Рисунок 9 – Проверка операции цикл for

Тест №9.

Состав теста: Проверка операций считывания с консоли и вывода в консоль.

Ожидаемый результат: корректные считывание данных из консоли и их вывод.

Входной текст программы:

```
var n, a, b, c, d, i, j : integer;
begin
    read(a);
    write(a);
end
```

Результат: в консоль корректно выведены корректно считанные данные. (Рисунок 10).

```
Input int value to assign variable with name "a": 100
Value of variable with name "a" is 100
```

Рисунок 10 – Проверка операций считывания с консоли и вывода в консоль

Тест №10.

Состав теста: Проверка вычисления выражения.

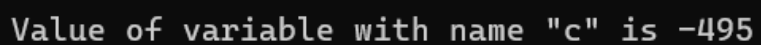
Ожидаемый результат: Вывод в консоль верного значения выражения.

Входной текст программы:

```
var n, a, b, c, d, i, j : integer;  
begin  
    a = (!5*8*(2+a)+(7-5+b-9*(5*5)+4));  
    write(a);  
end
```

Результат: в консоль выведено корректное значение выражения.

(Рисунок 11).



Value of variable with name "c" is -495

Рисунок 11 – Проверка вычисления выражения

Заключение

В ходе выполнения курсового проектирования по теме “Разработка транслятора в форме интерпретатора” была создана программа, реализующая транслятор с языка, определенного соответствующей формальной грамматике. В процессе были выполнены поставленные задачи, а именно: составлены требования к программному средству; создан детерминированный нисходящий автомат; создан транслятор-интерпретатор; реализована и протестирована программа.

В итоге создано приложение, которое выполняет лексический анализ исходного кода, синтаксический анализ исходного кода, трансляцию-интерпретацию исходного кода, предоставление возможности ввода и вывода данных.

В процессе разработки были изучены теория формальных грамматик, нисходящие детерминированные распознаватели и методы трансляции, а также получен опыт в создании и отладке программы на языке C# в Visual Studio 2022.

Список использованных источников

1. Синтаксический анализ. [Электронный ресурс]. – <https://studfile.net/preview/6062741/page:2/> – (дата обращения: 15.12.22)
2. Нисходящий метод. [Электронный ресурс]. – <https://studfile.net/preview/4349789/page:13/> – (дата обращения: 15.12.22)
3. Нисходящие распознаватели, LL(K) и разделенные грамматики. Построение распознавателя. [Электронный ресурс]. – <https://studfile.net/preview/1158226/page:8/> – (дата обращения: 15.12.22)
4. Метод рекурсивного спуска [Электронный ресурс]. – <https://studfile.net/preview/11218863/page:2/> – (дата обращения: 15.12.22)

Приложение А. Листинг программы

Файл Tokens.cs

```
namespace TheoryOfProgrammingLanguages.Tokens
{
    public enum TokenType
    {
        //symbols, delimiters
        IGNORE,
        SEMICOLON,
        COLON,
        COMMA,
        OPEN_PARENTHESIS,
        CLOSE_PARENTHESIS,
        //literals
        NUMBER,
        //keywords
        BEGIN,
        VAR,
        END,
        INTEGER,
        READ,
        WRITE,
        FOR,
        TO,
        DO,
        END_FOR,
        //variables
        IDENTIFIER,
        //operators
        ADDITION_OPERATOR,
        SUBTRACTION_OPERATOR,
        MULTIPLICATION_OPERATOR,
        DIVISION_OPERATOR,
        NEGATE_OPERATOR,
        ASSIGNMENT_OPERATOR
    }

    public class Token : IComparable<Token>
    {
```

```

public TokenType Type { get; }

public Token(TokenType type) => Type = type;

override

public string ToString() => $"type: {Type}";

//supposed to work only with operators tokens +, -, /, *, =

public int CompareTo(Token other)
{
    if(!IsOperator(other))

        throw new UnexpectedTokenException("Impossible to compare tokens. Only
operators can be compared");

    switch (Type)
    {
        case TokenType.ASSIGNMENT_OPERATOR:

            return 1;

        case TokenType.NEGATE_OPERATOR:

            if (other.Type == TokenType.ASSIGNMENT_OPERATOR)

                return -1;

            if (other.Type == TokenType.NEGATE_OPERATOR)

                return 0;

            return 1;

        case TokenType.DIVISION_OPERATOR:

        case TokenType.MULTIPLICATION_OPERATOR:

            if (other.Type == TokenType.DIVISION_OPERATOR || other.Type ==
TokenType.MULTIPLICATION_OPERATOR)

                return 0;

            if (other.Type==TokenType.NEGATE_OPERATOR || other.Type ==
TokenType.ASSIGNMENT_OPERATOR)

                return -1;

            return 1;

        case TokenType.ADDITION_OPERATOR:

        case TokenType.SUBTRACTION_OPERATOR:

            if (other.Type == TokenType.ADDITION_OPERATOR || other.Type ==
TokenType.SUBTRACTION_OPERATOR)

                return 0;

            return -1;

        default:

            throw new UnexpectedTokenException("Impossible to compare tokens.
Only operators can be compared");

    }
}

```

```

public static bool IsOperator(Token token)
{
    return token.Type == TokenType.ASSIGNMENT_OPERATOR ||
           token.Type == TokenType.ADDITION_OPERATOR ||
           token.Type == TokenType.SUBTRACTION_OPERATOR ||
           token.Type == TokenType.DIVISION_OPERATOR ||
           token.Type == TokenType.MULTIPLICATION_OPERATOR ||
           token.Type == TokenType.NEGATE_OPERATOR;
}

}

public class ValuedToken : Token
{
    public string Value { get; }

    public ValuedToken(TokenType type, string value) : base(type) => Value =
value;

    override
    public string ToString() => $"type: {Type}, value: {Value}";
}

public struct TokenDefinition
{
    public Regex Reg { get; }

    public TokenType Type { get; }

    public TokenDefinition(string pattern, TokenType type)
    {
        Reg = new Regex(pattern);

        Type = type;
    }

    public bool IsValuedToken() => Type == TokenType.IDENTIFIER || Type ==
TokenType.NUMBER;

    public bool IsIgnoredToken() => Type == TokenType.IGNORE;
}

}

```

Файл **Lexer.cs**

```

namespace TheoryOfProgrammingLanguages
{
    public class Lexer

```

```

{
    private string _input;
    private List<TokenDefinition> _tokenDefinitions;
    private int _cursorPosition;
    public Lexer(List<TokenDefinition> tokenDefinitions, string input)
    {
        _tokenDefinitions = tokenDefinitions;
        _input = input;
        _cursorPosition = 0;
    }
    private bool IsEndOfInput() => _cursorPosition >= _input.Length;
    public Queue<Token> GetTokens()
    {
        Queue<Token> tokens = new();
        while (!IsEndOfInput())
        {
            var token = GetNextToken();
            if (token != null)
            {
                if (token.Type == TokenType.IDENTIFIER)
                    CheckLengthOfIdentifier((ValuedToken)token);
                tokens.Enqueue(token);
            }
        }
        return tokens;
    }
    private void CheckLengthOfIdentifier(ValuedToken token)
    {
        if (token.Value.Length > 12)
            throw new IllegalTokenException("Max length of identifier is
12 symbols");
    }
    private Token? GetNextToken()
    {
        string target = _input.Substring(_cursorPosition);
        foreach (var tokenDefinition in _tokenDefinitions)

```



```

        {
            var matchResult = tokenDefinition.Reg.Match(target);
            if (!matchResult.Success)
                continue;
            _cursorPosition += matchResult.Value.Length;
            if (tokenDefinition.IsIgnoredToken())
                return null;
            if (tokenDefinition.IsValuedToken())
                return new ValuedToken(tokenDefinition.Type,
matchResult.Value);
            else
                return new Token(tokenDefinition.Type);
        }
        //if there is no match to token
        throw new IllegalTokenException($"Illegal token at the input:
{target}");
    }
}

```

Файл Parser.cs

```

namespace TheoryOfProgrammingLanguages
{
    internal class Parser
    {
        private Queue<Token> _tokens;

        public Parser(Queue<Token> tokens) => _tokens = tokens;

        private bool HasTokens() => _tokens.Count > 0;

        private Token ConsumeExpectedCurrentToken(TokenType type)
        {
            if (!HasTokens())
                throw new UnexpectedEndOfProgramException($"Unexpected end of
program, expected next token with type: \"{type}\"");
            var token = _tokens.Dequeue();
            if (token.Type != type)
                throw new UnexpectedTokenException($"Unexpected token:
\"{token}\", expected token with type: \"{type}\"");
            return token;
        }
    }
}

```

```

    }

    private bool TryConsumeExpectedCurrentToken(TokenType type)
    {
        if (!HasTokens())
            throw new UnexpectedEndOfProgramException($"Unexpected end of
program, expected next token with type: \"{type}\"");
        if (_tokens.Peek().Type == type)
        {
            _tokens.Dequeue();
            return true;
        }
        else
            return false;
    }

    private Token PeekCurrentToken()
    {
        if (!HasTokens())
            throw new UnexpectedEndOfProgramException($"Unexpected end of
program, expected existing next token");
        return _tokens.Peek();
    }

    public void ParseProgram(out List<ValuedToken> variables, out
List<IStatement> statements)
    {
        variables = ParseDeclaration();
        ConsumeExpectedCurrentToken(TokenType.BEGIN);
        statements = ParseInternalStatements(TokenType.END);
        ConsumeExpectedCurrentToken(TokenType.END);
        if (HasTokens())
            throw new UnexpectedEndOfProgramException($"Unexpected end of
program. There is symbol after {TokenType.END}: \"{PeekCurrentToken()}\"");
    }

    private List<ValuedToken> ParseDeclaration()
    {
        ConsumeExpectedCurrentToken(TokenType.VAR);
        var variables = GetVariablesList();
        ConsumeExpectedCurrentToken(TokenType.COLON);
    }

```

```

        ConsumeExpectedCurrentToken(TokenType.INTEGER);
        ConsumeExpectedCurrentToken(TokenType.SEMICOLON);
        return variables;
    }

    private List<ValuedToken> GetVariablesList()
    {
        List<ValuedToken> variables = new();
        do

variables.Add((ValuedToken) ConsumeExpectedCurrentToken(TokenType.IDENTIFIER))
;

        while (TryConsumeExpectedCurrentToken(TokenType.COMMA));
        return variables;
    }

    public List<IStatement> ParseInternalStatements(TokenType
endTokenType)
    {
        List<IStatement> statements = new List<IStatement>();
        while (PeekCurrentToken().Type != endTokenType)
            statements.Add(GetNextStatement());
        return statements;
    }

    private IStatement GetNextStatement()
    {
        switch (PeekCurrentToken().Type)
        {
            case TokenType.READ:
                return ParseReadStatement();

            case TokenType.WRITE:
                return ParseWriteStatement();

            case TokenType.FOR:
                return ParseForStatement();

            case TokenType.IDENTIFIER:

```

```

        return ParseAssignmentStatement();

        default:
            throw new UnexpectedTokenException($"Unexpected token
during executing statements: \"{PeekCurrentToken()}\"");
    }
}

private ForStatement ParseForStatement()
{
    ConsumeExpectedCurrentToken(TokenType.FOR);

    var iteratorVariable =
(ValuedToken) ConsumeExpectedCurrentToken(TokenType.IDENTIFIER);
    ConsumeExpectedCurrentToken(TokenType.ASSIGNMENT_OPERATOR);
    var startIteratorValueExpression = new List<Token>();
    GetInfixExpression(startIteratorValueExpression, TokenType.TO);
    ConsumeExpectedCurrentToken(TokenType.TO);

    List<Token> endIteratorValueInfixExpression = new();
    GetInfixExpression(endIteratorValueInfixExpression,
TokenType.DO);
    ConsumeExpectedCurrentToken(TokenType.DO);

    var statements = ParseInternalStatements(TokenType.END_FOR);
    ConsumeExpectedCurrentToken(TokenType.END_FOR);

    return new ForStatement(iteratorVariable,
startIteratorValueExpression, endIteratorValueInfixExpression, statements);
}

private ReadStatement ParseReadStatement()
{
    ConsumeExpectedCurrentToken(TokenType.READ);
    ConsumeExpectedCurrentToken(TokenType.OPEN_PARENTHESIS);
    ReadStatement readStatement = new( GetVariablesList() );
    ConsumeExpectedCurrentToken(TokenType.CLOSE_PARENTHESIS);
    ConsumeExpectedCurrentToken(TokenType.SEMICOLON);
    return readStatement;
}

```

```

    }

    private WriteStatement ParseWriteStatement()
    {
        ConsumeExpectedCurrentToken(TokenType.WRITE);
        ConsumeExpectedCurrentToken(TokenType.OPEN_PARENTHESIS);
        WriteStatement writeStatement = new( GetVariablesList() );
        ConsumeExpectedCurrentToken(TokenType.CLOSE_PARENTHESIS);
        ConsumeExpectedCurrentToken(TokenType.SEMICOLON);
        return writeStatement;
    }

    public AssignmentStatement ParseAssignmentStatement()
    {
        var variable =
        (ValuedToken) ConsumeExpectedCurrentToken(TokenType.IDENTIFIER);
        ConsumeExpectedCurrentToken(TokenType.ASSIGNMENT_OPERATOR);
        List<Token> valueExpression = new();
        GetInfixExpression(valueExpression, TokenType.SEMICOLON);
        ConsumeExpectedCurrentToken(TokenType.SEMICOLON);
        return new AssignmentStatement(variable, valueExpression);
    }

    private void GetInfixExpression(List<Token> infixExpression,
    TokenType endTokenType)
    {
        Token? prevExpressionToken = null;
        while (PeekCurrentToken().Type != endTokenType)
        {
            var currentExpressionToken =
            ConsumeExpectedCurrentToken(PeekCurrentToken().Type);

            if (currentExpressionToken.Type ==
            TokenType.OPEN_PARENTHESIS)
            {
                infixExpression.Add(currentExpressionToken);

                GetInfixExpression(infixExpression,
                TokenType.CLOSE_PARENTHESIS);

                prevExpressionToken =
                ConsumeExpectedCurrentToken(TokenType.CLOSE_PARENTHESIS);

                infixExpression.Add(prevExpressionToken);
                continue;
            }
        }
    }

```

```

        }

        if
(ExpressionHelper.IsTokenValidToInfixExpression(prevExpressionToken,
currentExpressionToken))

        {

            prevExpressionToken = currentExpressionToken;

            infixExpression.Add(prevExpressionToken);

        }

        else

            throw new UnexpectedTokenException($"Unexpected token in
the expression: \"{currentExpressionToken}\"");

    }

    if
(!ExpressionHelper.IsEndOfInfixExpressionValid(prevExpressionToken))

        throw new Exception($"Unexpected end of expression/internal
expression \"{prevExpressionToken}\" after \"{PeekCurrentToken()}\"");

    }

}

```

Файл Statements.cs

```

namespace TheoryOfProgrammingLanguages.Statements
{

    internal enum StatementType

    {

        READ_STATEMENT,

        WRITE_STATEMENT,

        FOR_STATEMENT,

        ASSIGNMENT_STATEMENT

    }

    internal interface IStatement

    {

        public StatementType Type { get; }

    }

    internal class ReadStatement : IStatement

    {

        public StatementType Type { get; }

    }

}

```

```

    public List<ValuedToken> Variables { get; }

    public ReadStatement(List<ValuedToken> variables)
    {
        Type = StatementType.READ_STATEMENT;
        Variables = variables;
    }
}

internal class WriteStatement : IStatement
{
    public StatementType Type { get; }
    public List<ValuedToken> Variables { get; }
    public WriteStatement(List<ValuedToken> variables)
    {
        Type = StatementType.WRITE_STATEMENT;
        Variables = variables;
    }
}

internal class ForStatement : IStatement
{
    public StatementType Type { get; }
    public AssignmentStatement IteratorStatement { get; }
    public List<Token> EndIteratorValueExpression { get; }
    public List<IStatement> Statements { get; }

    public ForStatement(ValuedToken iteratorVariable, List<Token>
startIteratorValueExpression, List<Token> endIteratorValueExpression,
List<IStatement> statements)
    {
        Type = StatementType.FOR_STATEMENT;
        IteratorStatement = new AssignmentStatement(iteratorVariable,
startIteratorValueExpression);
        EndIteratorValueExpression = endIteratorValueExpression;
        Statements = statements;
    }
}

internal class AssignmentStatement : IStatement
{
    public StatementType Type { get; }

```

```

    public ValuedToken Variable { get; }

    public List<Token> ValueInfixExpression { get; }

    public AssignmentStatement(ValuedToken variable, List<Token>
valueInfixExpression)
    {
        Type = StatementType.ASSIGNMENT_STATEMENT;
        Variable = variable;
        ValueInfixExpression = valueInfixExpression;
    }
}
}

```

Файл Interpreter.cs

```

namespace TheoryOfProgrammingLanguages
{

    internal class Interpreter
    {
        private Dictionary<string, int?> _variables;

        public Interpreter() => _variables= new Dictionary<string, int?>();

        private bool IsVariableDefined(ValuedToken variable) =>
_variables.ContainsKey(variable.Value);

        private bool IsVariableAssigned(ValuedToken variable) =>
_variables[variable.Value] != null;

        private void DefineVariable(ValuedToken variable)
        {
            if (!_variables.TryAdd(variable.Value, null))
                throw new InvalidOperationException($"The variable with name
{variable.Value} already defined");
        }

        private void AssignVariable(ValuedToken variable, int value)
        {
            if (!IsVariableDefined(variable))
                throw new InvalidVariableException($"The variable with name
{variable.Value} is not defined");
            _variables[variable.Value] = value;
        }

        private int GetVariableValue(ValuedToken variable)

```



```

    {
        if (!IsVariableDefined(variable))
            throw new InvalidVariableException($"The variable with name
{variable.Value} is not defined");
        if (!IsVariableAssigned(variable))
            throw new InvalidVariableException($"The variable with name
{variable.Value} is not assigned");
        return (int)_variables[variable.Value];
    }

    public void Interpretate(List<ValuedToken> variables,
List<IStatement> statements)
    {
        foreach (ValuedToken variable in variables)
            DefineVariable(variable);
        ExecuteInternalStatements(statements);
    }
    private void ExecuteInternalStatements(List<IStatement> statements)
    {
        foreach(IStatement statement in statements)
            ExecuteStatement(statement);
    }
    private void ExecuteStatement(IStatement statement)
    {
        switch (statement.Type)
        {
            case StatementType.WRITE_STATEMENT:
                ExecuteWriteStatement((WriteStatement)statement);
                break;
            case StatementType.READ_STATEMENT:
                ExecuteReadStatement((ReadStatement)statement);
                break;
            case StatementType.FOR_STATEMENT:
                ExecuteForStatement((ForStatement)statement);
                break;
            case StatementType.ASSIGNMENT_STATEMENT:

```

```

ExecuteAssignmentStatement((AssignmentStatement) statement);

        break;

    default:

        throw new InvalidOperationException("Illegal statement in
the program");
    }

}

private void ExecuteWriteStatement(WriteStatement statement)
{
    foreach (var variable in statement.Variables)
    {
        if (IsVariableAssigned(variable))

            Console.WriteLine($"Value of variable with name
\"{variable.Value}\" is {_variables[variable.Value]}");

        else

            throw new InvalidVariableException($"The variable with
name \"{variable.Value}\" is not assigned");

    }

}

private void ExecuteReadStatement(ReadStatement statement)
{
    foreach (var variable in statement.Variables)
    {

        Console.Write($"Input int value to assign variable with name
\"{variable.Value}\": ");

        var value = Console.ReadLine();

        AssignVariable(variable, ExpressionHelper.ParseToInt(value));

    }

}

private void ExecuteAssignmentStatement(AssignmentStatement
statement)
{
    var value =
ExpressionHelper.EvaluateInfixExpressionWithVariables(statement.ValueInfixExp
ression, GetVariableValue);

    AssignVariable(statement.Variable, value);

}

private void ExecuteForStatement(ForStatement statement)

```

```

    {
        ExecuteAssignmentStatement(statement.IteratorStatement);
        var iterator = statement.IteratorStatement.Variable;
        int maxIteratorValue =
ExpressionHelper.EvaluateInfixExpressionWithVariables(statement.EndIteratorVa
lueExpression, GetVariableValue);
        while ( GetVariableValue(iterator) < maxIteratorValue)
        {
            ExecuteInternalStatements(statement.Statements);
            AssignVariable(iterator, GetVariableValue(iterator) + 1);
        }
    }
}
}

```

Файл ExpressionHelper.cs

```

namespace TheoryOfProgrammingLanguages.Expression
{
    internal class ExpressionHelper
    {
        public delegate int GetVariableValue(ValuedToken variable);

        private static bool CanAddBinaryOperatorToInfixExpression(Token?
prevToken)
        {
            //allowed after IDENTIFIER, NUMBER, )
            return prevToken != null &&
                (prevToken.Type == TokenType.IDENTIFIER ||
                 prevToken.Type == TokenType.NUMBER ||
                 prevToken.Type == TokenType.CLOSE_PARENTHESIS);
        }

        private static bool CanAddUnaryOperatorToInfixExpression(Token?
prevToken)
        {
            //allowed in the begin and after +, -, /, *, (
            return prevToken == null ||
                prevToken.Type == TokenType.ADDITION_OPERATOR ||

```

```

        prevToken.Type == TokenType.SUBTRACTION_OPERATOR ||
        prevToken.Type == TokenType.MULTIPLICATION_OPERATOR ||
        prevToken.Type == TokenType.DIVISION_OPERATOR ||
        prevToken.Type == TokenType.OPEN_PARENTHESIS;
    }

    private static bool CanAddOperandToInfixExpression(Token? prevToken)
    {
        //allowed in the begin and after +, -, /, *, (, !
        return prevToken == null ||
            (prevToken.Type != TokenType.IDENTIFIER &&
            prevToken.Type != TokenType.NUMBER &&
            prevToken.Type != TokenType.CLOSE_PARENTHESIS);
    }

    public static int ParseToInt(string value) => Int32.Parse(value);
    public static bool IsEndOfInfixExpressionValid(Token? endToken)
    {
        //allowed end with IDENTIFIER, NUMBER, (
        return endToken != null &&
            (endToken.Type == TokenType.IDENTIFIER ||
            endToken.Type == TokenType.NUMBER ||
            endToken.Type == TokenType.CLOSE_PARENTHESIS);
    }

    public static bool IsTokenValidToInfixExpression(Token? prevToken,
    Token currentToken)
    {
        switch (currentToken.Type)
        {
            case TokenType.IDENTIFIER:
            case TokenType.NUMBER:
                return CanAddOperandToInfixExpression(prevToken);
            case TokenType.ADDITION_OPERATOR:
            case TokenType.SUBTRACTION_OPERATOR:
            case TokenType.DIVISION_OPERATOR:
            case TokenType.MULTIPLICATION_OPERATOR:
                return CanAddBinaryOperatorToInfixExpression(prevToken);
        }
    }

```

```

        case TokenType.NEGATE_OPERATOR:

            return CanAddUnaryOperatorToInfixExpression(prevToken);

        default:

            throw new UnexpectedTokenException($"Unexpected token in
the expression: \"{currentToken}\" after token \"{ prevToken}\"");

    }

}

private static bool IsEndOfInternalExpressionOperators(Stack<Token>
operators)

{

    return operators.Count != 0 && operators.Peek().Type !=
TokenType.OPEN_PARENTHESIS;

}

private static List<Token> GetPostfixExpression(List<Token>
infixExpression)

{

    List<Token> postfixExpression = new();

    Stack<Token> operators = new();

    foreach (Token token in infixExpression)

    {

        if (Token.IsOperator(token))

        {

            while (IsEndOfInternalExpressionOperators(operators) &&
operators.Peek().CompareTo(token) >=0 )

                postfixExpression.Add(operators.Pop());

            operators.Push(token);

            continue;

        }

        switch (token.Type)

        {

            case TokenType.NUMBER:

            case TokenType.IDENTIFIER:

                postfixExpression.Add(token);

                break;

            case TokenType.OPEN_PARENTHESIS:

                operators.Push(token);

```

```

        break;

        case TokenType.CLOSE_PARENTHESIS:
            while (IsEndOfInternalExpressionOperators(operators))
                postfixExpression.Add(operators.Pop());
            operators.Pop();
            break;

        default:
            throw new UnexpectedTokenException($"Undefined token
in the expression: {token}");
    }
}

while (operators.Count != 0)
    postfixExpression.Add(operators.Pop());
return postfixExpression;
}

public static int EvaluateInfixExpressionWithVariables(List<Token>
infixExpression, GetVariableValue getVariableValue)
{
    Stack<int> operands = new();
    var postfixExpression = GetPostfixExpression(infixExpression);
    foreach (Token token in postfixExpression)
    {
        switch (token.Type)
        {
            case TokenType.ADDITION_OPERATOR:
            case TokenType.SUBTRACTION_OPERATOR:
            case TokenType.MULTIPLICATION_OPERATOR:
            case TokenType.DIVISION_OPERATOR:
            case TokenType.NEGATE_OPERATOR:
                operands.Push(GetResultOfCurrentOperation(token,
operands));
                break;
            case TokenType.NUMBER:
                operands.Push(ParseToInt(((ValuedToken) token).Value));
                break;

```

```

        case TokenType.IDENTIFIER:
            operands.Push(getVariableValue((ValuedToken) token));
            break;

        default:
            throw new UnexpectedTokenException($"Undefined token
in the expression: {token}");
    }
}

return operands.Pop();
}

private static int GetResultOfCurrentOperation(Token operatorToken,
Stack<int> operands)
{
    int left;
    int right;
    switch (operatorToken.Type)
    {
        case TokenType.ADDITION_OPERATOR:
            return operands.Pop() + operands.Pop();
        case TokenType.SUBTRACTION_OPERATOR:
            right = operands.Pop();
            left = operands.Pop();
            return left - right;
        case TokenType.MULTIPLICATION_OPERATOR:
            return operands.Pop() * operands.Pop();
        case TokenType.DIVISION_OPERATOR:
            right = operands.Pop();
            left = operands.Pop();
            return left / right;
        case TokenType.NEGATE_OPERATOR:
            return -operands.Pop();
        default:
            throw new UnexpectedTokenException($"Undefined operator
in the expression: {operatorToken}");
    }
}

```

```
    }  
  }  
}
```

Приложение Б. Диаграмма классов

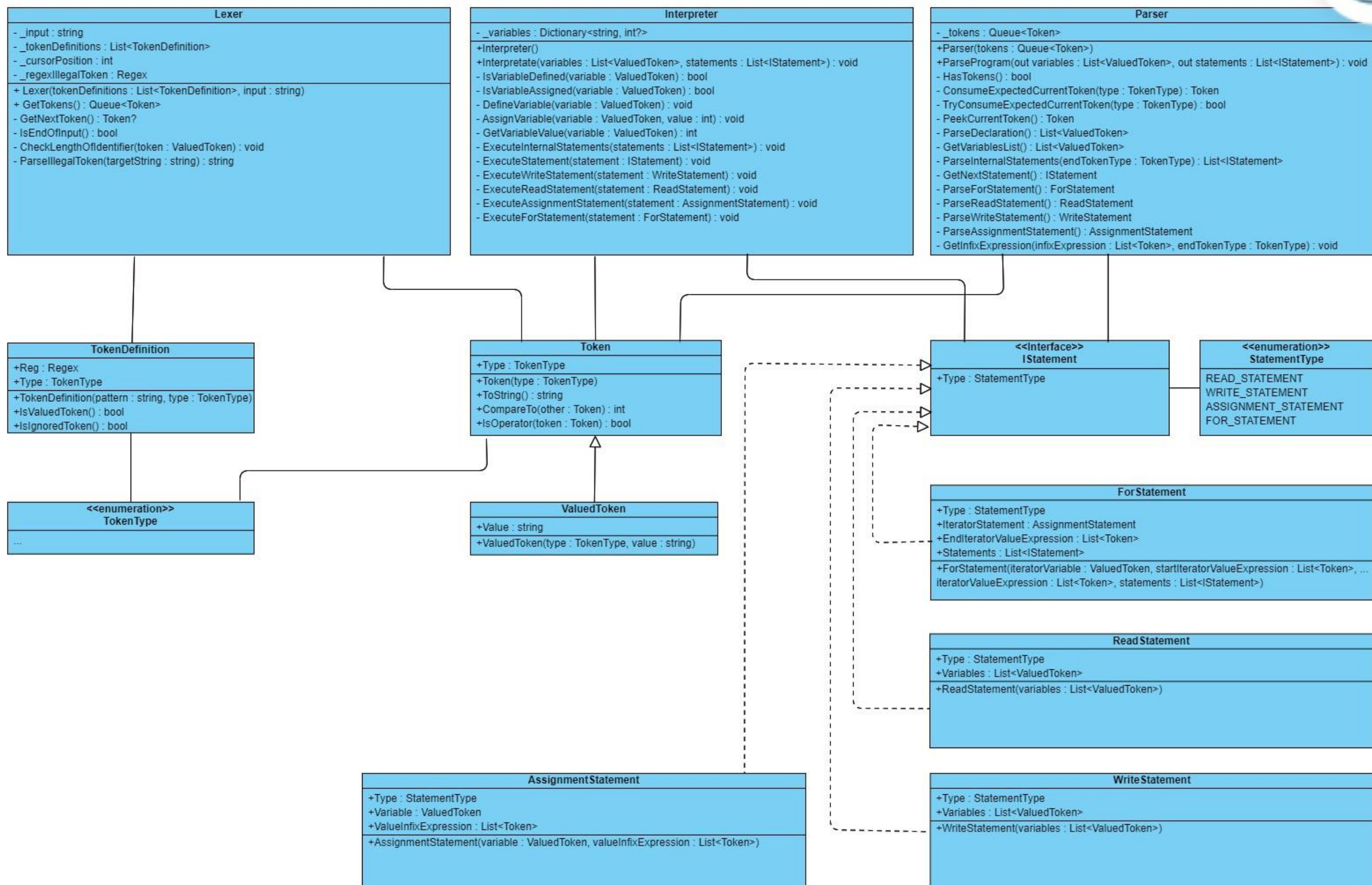


Рисунок 1. Диаграмма классов