

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ (ШКОЛА)**

**Департамент программной инженерии и искусственного интеллекта**

РАЗРАБОТКА ТРАНСЛЯТОРА, ПЕРЕВОДЯЩЕГО ПОДМНОЖЕСТВО ЯЗЫКА KOTLIN В ЭКВИВАЛЕНТНОЕ ПОДМНОЖЕСТВО ЯЗЫКА C#

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Теория языков программирования и компиляторы»

по образовательной программе подготовки бакалавров по направлению 09.03.04 Программная инженерия

Выполнили студенты

гр. Б9120-09.03.04прогин

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Воронова Д.В.,

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мельникова Е.А.

Руководитель:

Глазырин Матвей Александрович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

г. Владивосток

2024 г

**Оглавление**

[Введение 4](#_Toc156856595)

[1 Неформальная постановка задачи 5](#_Toc156856596)

[2 Синтаксис входного языка 6](#_Toc156856597)

[3 Контекстные условия подмножества языка Kotlin 14](#_Toc156856598)

[5 Таблица соответствия языков 18](#_Toc156856599)

[6 Проект лексического анализатора 24](#_Toc156856600)

[6.1 Модель данных 24](#_Toc156856601)

[6.2 Таблица ключевых слов 24](#_Toc156856602)

[6.3 Таблица зарезервированных имен 25](#_Toc156856603)

[6.4 Типы лексем 27](#_Toc156856604)

[6.5 Типы ошибок 28](#_Toc156856605)

[6.6 Конечный автомат лексического анализатора 29](#_Toc156856606)

[7 Проект семантического анализатора 32](#_Toc156856607)

[7.1 Модель данных 32](#_Toc156856608)

[7.2 Описание алгоритма рекурсивного спуска 32](#_Toc156856609)

[7.3 Пример дерева разбора 33](#_Toc156856610)

[7.4 Типы ошибок 33](#_Toc156856611)

[8 Проект синтаксическогоо анализатора 36](#_Toc156856612)

[8.1 Модель данных 36](#_Toc156856613)

[8.2 Методы проверки нарушений контекстных условия 36](#_Toc156856614)

[8.3 Типы ошибок 36](#_Toc156856615)

[9 Проект генератора кода 38](#_Toc156856616)

[9.1 Модель данных 38](#_Toc156856617)

[9.2 Метод кодогенерации 38](#_Toc156856618)

[9.3 Примеры кодогенерации 39](#_Toc156856619)

[10 Проект оптимизатора кода 40](#_Toc156856620)

[10.1 Модель данных 40](#_Toc156856621)

[10.2 Методы оптимизации кода 40](#_Toc156856622)

[10.3 Примеры оптимизации кода 40](#_Toc156856623)

[11 Тестирование 42](#_Toc156856624)

[11.1 Лексический анализатор 42](#_Toc156856625)

[11.2 Синтаксический анализатор 42](#_Toc156856626)

[11.3 Семантический анализатор 42](#_Toc156856627)

[11.4 Оптимизатор кода 42](#_Toc156856628)

[11.5 Генератор кода 42](#_Toc156856629)

[Заключение 45](#_Toc156856630)

# Введение

**Транслятор** — программа или техническое средство, преобразовывающее программу, представленную на одном из языков программирования, в программу, написанную на другом языке.

Язык программирования — это формальная знаковая система, на которой пишут компьютерную программу.

В данном курсовом проекте в качестве входного языка был выбран язык Kotlin, в качестве выходного – C#.

**Цель курсового проекта:** разработать проект транслятора, который преобразует программу, содержащую подмножество языка X\_LANG, в программу, содержащую подмножество языка Y\_LANG генерируя эквивалентный исходный код.

**Задачи:**

1. описать грамматику подмножества входного языка;
2. описать контекстные условия входного и выходного языков;
3. описать соответствие конструкций входного и выходного языков;
4. разработать проект лексического анализатора;
5. разработать проект синтаксического анализатора;
6. разработать проект семантического анализатора;
7. разработать проект генератора кода выходного языка;
8. реализовать транслятор входного языка в выходной;
9. выполнить тестирование разработанного транслятора.

# 1 Неформальная постановка задачи

В ходе данного курсового проекта необходимо разработать транслятор, который будет переводить подмножество языка Kotlin в подмножество языка C#. Для этого нужно проанализировать и описать входной и выходной язык, разработать лексический, синтаксический и семантический анализаторы, а также генератор кода выходного языка. На основе этих шагов нужно реализовать и протестировать транслятор.

Все переменные в языке Kotlin являются объектами в том смысле, что можно вызывать функции-члены и свойства любых переменных.

Некоторые типы могут иметь специальное внутреннее представление – например, числа, символы и логические значения могут быть представлены как примитивные значения во время выполнения, – но для пользователя они выглядят как обычные классы.

Таким образом, рассматриваемое подмножество языка Kotlin включает в себя такие типы данных, как Integer, Float, Double, Boolean, character, string, array. Integer включает в себя Byte, Short, Int, Long. Чтобы объявить переменную, тип данных которой не будет меняться, нужно использовать оператор val, и var в противном случае.

Массивы в языке создаются с помощью функции arrayOf() или синонимов этой функции. Узнать длину массива можно при помощи свойства size, а входящие в массив индексы при помощи indices. Перебор можно сделать с использованием цикла for или forEach.

Списки объявляются с помощью функции listOf(). С помощью функций add() и remove() в них можно добавлять или удалять элементы. Доступ к отдельному элементу списка можно получить по номеру индекса через квадратные скобки. Также можно использовать метод get() с указанием номера индекса.

В подмножестве языка Kotlin содержатся логические операции дизъюнкции, конъюнкции, отрицания и стандартные арифметические операции сложения, вычитания, умножения, деления и деления по модулю. Из унарных операций включается логическое отрицание.

В подмножестве Kotlin есть такие виды циклов, как for, while. Язык поддерживает традиционные операторы break и continue для выхода из них.

В подмножестве языка Kotlin есть два оператора ветвления: if для одиночного ветвления и when – для множественного. Оба ветвления могут иметь как блочную, так и строчную структуру.

Функции в Kotlin объявляются ключевым словом fun. Стандартными функциями являются различные математические функции, например abs(), ceil(), floor(), max(), min(), round(), sqrt(), cbrt(), exp(). log(), pow(), sign() и тригонометрические функции. Также к стандартным относятся функции для работы с вводом и выводом, то есть print(), println(), readLine().

# 2 Синтаксис входного языка

**Аксиома:**

program

**Правила:**

**program** ::=statement EOL EOF | statement EOL program | COMMENT EOL EOF | COMMENT EOL program

**statement** ::= declaration | assigning | function-call | function-construction | if-construction| while-construction | return-statement | switch-statement | for-statement | array-statement

**declaration** ::= simple-declaration | simple-declaration EQUALS expression | simple-declaration EQUALS term

**simple-declaration** ::= VAL ID | VAR ID

**value** ::= STR | CHAR | NUM | TRUE | FALSE | MINUS NUM

**expression** ::= expression-inside | LBR expression-inside RBR | LBR logical-expression RBR | logical-expression| LBR expression-inside RBR expression-inside’ | LBR logical-expression RBR logical-expression’ | logical-expression logical-expression’

**expression-inside** ::= LBR expression-inside RBR | term sign expression-inside | term sign term | LBR expression-inside RBR expression-inside’ | term sign expression-inside expression-inside’ | term sign term expression-inside’

**expression-inside’** ::= sign expression | sign term | sign expression expression-inside’ | sign term expression-inside’

**logical-expression** ::= LBR logical-expression RBR | NOT term | NOT logical-expression | term logical-sign logical-expression | term logical-sign term | ID IS type LBR logical-expression RBR logical-expression’ | NOT term logical-expression’ | NOT logical-expression logical-expression’ | expression-inside logical-expression’

**logical-expression’** ::= logical-sign term | logical-sign logical-expression | logical-sign term logical-expression’ | logical-sign logical-expression logical-expression’

**term** ::= value | ID | function-call | ID LSQBR NUM RSQBR| MINUS ID | MINUS ID LSQBR NUM RSQBR | MINUS function-call

**logical-sign** ::= MORE | LESS | MOREOREQUAL | LESSOREQUAL | AND | OR | ISEQUAL | ISNOTEQUAL

**sign** ::= PLUS | MINUS | MUL | MOD | DIV

**function-call** ::= ID LBR actual-params RBR | ID LBR RBR

**actual-params** ::= actual-param COMMA actual-params | actual-param

**actual-param** ::= expression | term

**assigning** ::= ID EQUALS expression | ID EQUALS term

**function-construction** ::= FUNCTION ID LBR params RBR LCBR function-block RCBR | FUNCTION ID LBR RBR LCBR function-block RCBR | FUNCTION ID LBR params RBR LCBR RCBR | FUNCTION ID LBR RBR LCBR RCBR | FUNCTION ID LBR RBR COLON type LCBR RCBR | FUNCTION ID LBR params RBR COLON type LCBR function-block RCBR | FUNCTION ID LBR RBR COLON type LCBR function-block RCBR

**params** ::= param COMMA params | param

**param** ::= ID COLON type

**type** ::= BYTE | SHORT | INT | LONG | FLOAT | DOUBLE | BOOL | CHARTYPE | STRING

**function-block** ::= EOL | EOL function-block | function-command function-block | function-command

**function-command** ::= declaration | assigning | function-call | if-construction| while-construction | switch-statement | return-statement| for-statement| array-statement

**if-construction** ::= IF LBR if-condition RBR LCBR function-block RCBR | IF LBR if-condition RBR LCBR RCBR | IF LBR if-condition RBR LCBR function-block RCBR ELSE LCBR function-block RCBR | IF LBR if-condition RBR LCBR RCBR ELSE LCBR function-block RCBR | IF LBR if-condition RBR LCBR function-block RCBR ELSE LCBR RCBR | IF LBR if-condition RBR LCBR RCBR ELSE LCBR RCBR

**if-condition** ::= logical-expression | term

**while-construction** ::= WHILE LBR if-condition RBR LCBR function-block RCBR | WHILE LBR if-condition RBR EOL LCBR function-block RCBR | WHILE LBR if-condition RBR LCBR EOL function-block RCBR

**return-statement** ::= RETURN term | RETURN expression

**switch-statement** ::= WHEN LBR ID RBR LCBR switch-block RCBR

**switch-block** ::= switch-command | switch-command EOL | switch-command EOL switch-block |EOL switch-command

**switch-command** ::= value CASE LCBR function-block RCBR | value CASE LCBR RCBR | ELSE CASE LCBR function-block RCBR | ELSE CASE LCBR RCBR | value CASE function-command | ELSE CASE function-command

**for-statement** ::= FOR LBR ID IN ID RBR LCBR for-block RCBR | FOR LBR ID IN ID RBR LCBR RCBR | FOR LBR ID IN NUM TO NUM RBR LCBR for-block RCBR | FOR LBR ID IN NUM TO NUM RBR LCBR RCBR | FOR LBR ID IN ID RBR for-statement | FOR LBR ID IN NUM TO NUM RBR for-statement

**for-block** ::= EOL | EOL for-block | for-command for-block | for-command

**for-command** ::= declaration | assigning | function-call | if-construction| while-construction | switch-statement | return-statement | CONTINUE | BREAK

**array-statement** ::= ID POINT ADD LBR value RBR | ID POINT REMOVE LBR NUM RBR

**Обозначения**

Для подмножества языка Kotlin введены следующие обозначения:

*RAWSTRING* – значение типа Raw strings,

*TO* – оператор "..",

*COMMENT* – комментарий "/\*\*/",

*COMMA* – запятая ",",

*LBR* – левая круглая скобка "(",

*RBR* – правая круглая скобка ")",

*BREAK* – ключевое слово"break",

*ABS* – математическая функция "abs()",

*CEIL* – математическая функция "ceil()",

*FLOOR* – математическая функция "floor()",

*MAX* – математическая функция "max()",

*MIN* – математическая функция "min()",

*ROUND* – математическая функция "round()",

*SQRT* – математическая функция "sqrt()",

*CBRT* – математическая функция "cbrt()",

*EXP* – математическая функция "exp()",

*LOG* – математическая функция "log()",

*POW* – математическая функция "pow()",

*SIGN* – математическая функция "sign()",

*CONTINUE* – ключевое слово"continue",

*SIN* – математическая функция "sin()",

*COS* – математическая функция "cos()",

*TAN* – математическая функция "tan()",

*LCBR* – левая фигурная скобка "{",

*RCBR* – правая фигурная скобка "}",

*LSQBR* – левая квадратная скобка "[",

*RSQBR* – правая квадратная скобка "]",

*COLON* – двоеточие ":",

*NOT* – логический оператор "!",

*AND* – логический оператор "&&",

*OR* – логический оператор "||",

*NUM* – численное значение, может быть целым или дробным,

*IN* – ключевое слово"in",

*PLUS* – арифметическая операция "+",

*CASE* – оператор "->",

*MINUS* – арифметическая операция "-",

*MUL* – арифметическая операция "\*",

*MOD* – арифметическая операция "%",

*DIV* – арифметическая операция "/",

*VAR* – ключевое слово"var",

*VAL* – ключевое слово"val",

*ISEQUAL* – логический оператор "==",

*ISNOTEQUAL* – логический оператор "!==",

*EQUALS* – оператор присваивания "=",

*IF* – ключевое слово"if",

*ELSE* – ключевое слово "else",

*WHEN* – ключевое слово"when",

WHILE – ключевое слово"while",

*FOR* – ключевое слово"for",

*MOREOREQUAL* – логический оператор ">=",

*LESSOREQUAL* – логический оператор "<=",

*MORE* – логический оператор ">",

*LESS*– логический оператор "<",

*READLINE* – стандартная функция "readLine()",

*IS* – ключевое слово"is"),

*FUNCTION* – ключевое слово"fun",

*RETURN* – ключевое слово"return",

*BYTE* – обозначение типа "Byte",

*SHORT* – обозначение типа "Short",

*INT* – обозначение типа "Int",

*LONG* – обозначение типа "Long",

*FLOAT* – обозначение типа "Float",

*DOUBLE* – обозначение типа "Double",

*BOOL* – обозначение типа "Boolean",

*CHARTYPE* – обозначение типа "Char",

*STRING* – обозначение типа "String",

*TRUE* – значение "true" типа Boolean,

*FALSE* – значение "false" типа Boolean,

*CHAR* – значение типа Char,

*ARRAYOF* – стандартная функция "arrayOf()",

*LISTOF* – стандартная функция "listOf()",

*POINT* – точка ".",

*ADD* – стандартный метод для массива "add()",

*REMOVE* – стандартный метод для массива "remove()",

*STR* – значение типа String,

*PRINTLN* – стандартная функция "println()",

*PRINT* – стандартная функция "print()",

*ID* – идентификатор,

*EOL* – символ окончания строки "®",

*EOF* – символ окончания файла.

**Описание грамматики языка**

Аксиома:

*program*

Правила:

*program* ::= statement EOL EOF | statement EOL program | COMMENT EOL EOF | COMMENT EOL program

*statement* ::= declaration | assigning | function-call | function-construction | if-construction| while-construction | return-statement | switch-statement | for-statement | array-statement

*declaration* ::= simple-declaration | simple-declaration EQUALS expression | simple-declaration EQUALS term

*simple-declaration* ::= VAL ID | VAR ID

*value* ::= STR | CHAR | NUM | TRUE | FALSE | MINUS NUM

*expression* ::= expression-inside | LBR expression-inside RBR | LBR logical-expression RBR | logical-expression| LBR expression-inside RBR expression-inside’ | LBR logical-expression RBR logical-expression’ | logical-expression logical-expression’

*expression-inside* ::= LBR expression-inside RBR | term sign expression-inside | term sign term | LBR expression-inside RBR expression-inside’ | term sign expression-inside expression-inside’ | term sign term expression-inside’

*expression-inside’* ::= sign expression | sign term | sign expression expression-inside’ | sign term expression-inside’

*logical-expression* ::= LBR logical-expression RBR | NOT term | NOT logical-expression | term logical-sign logical-expression | term logical-sign term | ID IS type LBR logical-expression RBR logical-expression’ | NOT term logical-expression’ | NOT logical-expression logical-expression’ | expression-inside logical-expression’

*logical-expression’* ::= logical-sign term | logical-sign logical-expression | logical-sign term logical-expression’ | logical-sign logical-expression logical-expression’

*term* ::= value | ID | function-call | ID LSQBR NUM RSQBR| MINUS ID | MINUS ID LSQBR NUM RSQBR | MINUS function-call

*logical-sign* ::= MORE | LESS | MOREOREQUAL | LESSOREQUAL | AND | OR | ISEQUAL | ISNOTEQUAL

*sign* ::= PLUS | MINUS | MUL | MOD | DIV

*function-call* ::= ID LBR actual-params RBR | ID LBR RBR

*actual-params* ::= actual-param COMMA actual-params | actual-param

*actual-param* ::= expression | term

*assigning* ::= ID EQUALS expression | ID EQUALS term

*function-construction* ::= FUNCTION ID LBR params RBR LCBR function-block RCBR | FUNCTION ID LBR RBR LCBR function-block RCBR | FUNCTION ID LBR params RBR LCBR RCBR | FUNCTION ID LBR RBR LCBR RCBR | FUNCTION ID LBR RBR COLON type LCBR RCBR | FUNCTION ID LBR params RBR COLON type LCBR function-block RCBR | FUNCTION ID LBR RBR COLON type LCBR function-block RCBR

*params* ::= param COMMA params | param

*param* ::= ID COLON type

*type* ::= BYTE | SHORT | INT | LONG | FLOAT | DOUBLE | BOOL | CHARTYPE | STRING

*function-block* ::= EOL | EOL function-block | function-command function-block | function-command

*function-command* ::= declaration | assigning | function-call | if-construction| while-construction | switch-statement | return-statement| for-statement| array-statement

*if-construction* ::= IF LBR if-condition RBR LCBR function-block RCBR | IF LBR if-condition RBR LCBR RCBR | IF LBR if-condition RBR LCBR function-block RCBR ELSE LCBR function-block RCBR | IF LBR if-condition RBR LCBR RCBR ELSE LCBR function-block RCBR | IF LBR if-condition RBR LCBR function-block RCBR ELSE LCBR RCBR | IF LBR if-condition RBR LCBR RCBR ELSE LCBR RCBR

*if-condition* ::= logical-expression | term

*while-construction* ::= WHILE LBR if-condition RBR LCBR function-block RCBR | WHILE LBR if-condition RBR EOL LCBR function-block RCBR | WHILE LBR if-condition RBR LCBR EOL function-block RCBR

*return-statement* ::= RETURN term | RETURN expression

*switch-statement* ::= WHEN LBR ID RBR LCBR switch-block RCBR

*switch-block* ::= switch-command | switch-command EOL | switch-command EOL switch-block |EOL switch-command

*switch-command* ::= value CASE LCBR function-block RCBR | value CASE LCBR RCBR | ELSE CASE LCBR function-block RCBR | ELSE CASE LCBR RCBR | value CASE function-command | ELSE CASE function-command

*for-statement* ::= FOR LBR ID IN ID RBR LCBR for-block RCBR | FOR LBR ID IN ID RBR LCBR RCBR | FOR LBR ID IN NUM TO NUM RBR LCBR for-block RCBR | FOR LBR ID IN NUM TO NUM RBR LCBR RCBR | FOR LBR ID IN ID RBR for-statement | FOR LBR ID IN NUM TO NUM RBR for-statement

*for-block* ::= EOL | EOL for-block | for-command for-block | for-command

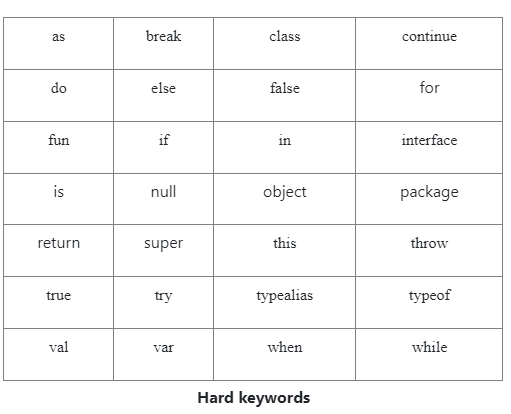
*for-command* ::= declaration | assigning | function-call | if-construction| while-construction | switch-statement | return-statement | CONTINUE | BREAK

*array-statement* ::= IDPOINT ADD LBR value RBR | ID POINT REMOVE LBR NUM RBR

# 3 Контекстные условия подмножества языка Kotlin

**Контекстные условия о правилах описания идентификаторов:**

* Длина идентификаторов не ограничена.
* Идентификатор может состоять из букв, цифр и знака нижнего подчеркивания, но начинаться может только с буквы или нижнего подчеркивания.
* Пробелы в названии идентификаторов недопустимы.
* Название идентификатора не может состоять только из знака нижнего подчеркивания или из нескольких таких знаков.
* Два идентификатора являются одинаковыми только в том случае, если они состоят из одних и тех же символов Юникода для каждого символа, соответственно, названия идентификаторов чувствительны к регистру.
* Идентификаторы не могут иметь такое же имя, как у зарезервированных слов, представленных на рисунке 1.



*Рисунок 1 – Зарезервированные слова в Kotlin*

**Условия соответствия между определяющими и использующими вхождениями идентификаторов:**

* Все используемые в программе идентификаторы должны быть инициализированы до их использования в программе.
* Идентификаторы могут быть объявлены без инициализации.
* Каждый из идентификаторов, используемых в формальном описании функции, должен быть описан только один раз.
* Идентификаторы типа val не могут быть инициализированы повторно.
* Каждый из идентификаторов, используемых в функции, должен быть описан только один раз.
* Каждый из идентификаторов, используемых в подпрограмме, должен быть описан только один раз.
* Каждый из идентификаторов, используемых в программе, должен быть описан один раз.
* Каждый из идентификаторов, используемых в блоке программе, должен быть описан один раз.
* Области видимости одинаковых идентификаторов не должны пересекаться.

**Условия соответствия видов значений:**

* Типы фактических и формальных параметров функций должны совпадать.
* Количество формальных и фактических параметров функции должно совпадать.

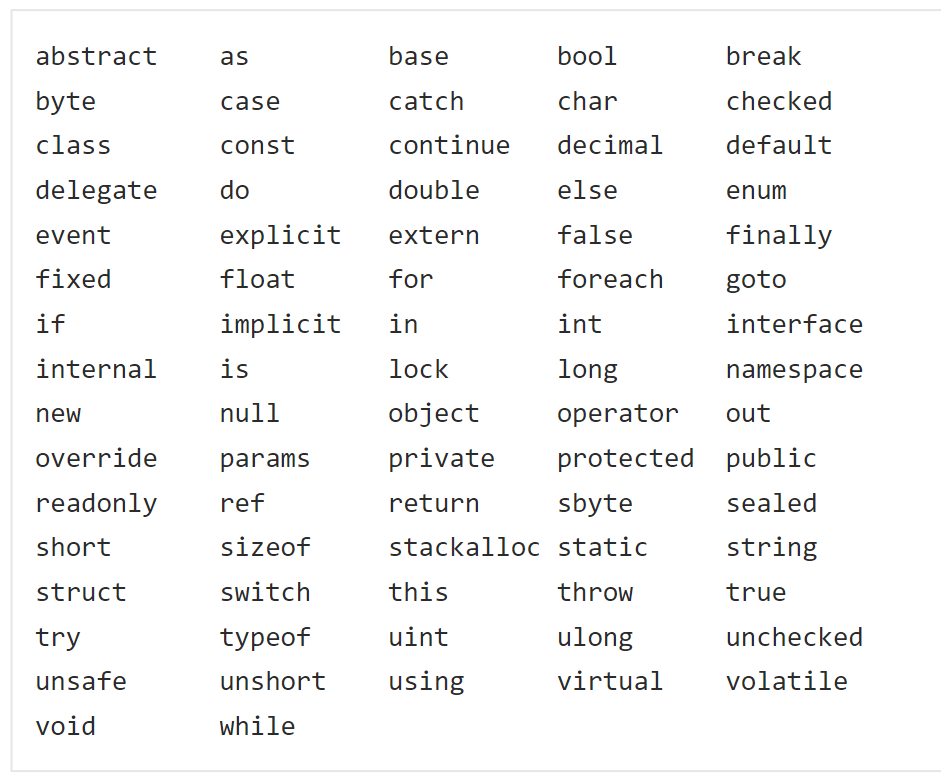
**Условия, определяющие количественные ограничения:**

* Количество формальных параметров функций не ограничено.
* Глубина вложенности блоков не ограничена.

**4 Контекстные условия языка С#**

**Контекстные условия о правилах описания идентификаторов:**

* Длина идентификаторов не ограничена.
* Идентификатор может состоять из букв, цифр и знака нижнего подчеркивания, но начинаться может только с буквы или нижнего подчеркивания.
* Пробелы в названии идентификаторов недопустимы.
* Название идентификатора не может состоять только из знака нижнего подчеркивания или из нескольких таких знаков.
* Два идентификатора являются одинаковыми только в том случае, если они состоят из одних и тех же символов Юникода для каждого символа, соответственно, названия идентификаторов чувствительны к регистру.
* Идентификаторы не могут иметь такое же имя, как у зарезервированных слов, представленных на рисунке 2.

****

*Рисунок 2 – Зарезервированные слова С#*

**Условия соответствия между определяющими и использующими вхождениями идентификаторов:**

* Все используемые в программе идентификаторы должны быть инициализированы до их использования в программе, должен быть указан их тип.
* Идентификаторы могут быть объявлены без инициализации с указанием типа.
* Каждый из идентификаторов, используемых в формальном описании функции, должен быть описан только один раз.
* Идентификаторы типа const не могут быть инициализированы повторно.
* Каждый из идентификаторов, используемых в функции, должен быть описан только один раз.
* Каждый из идентификаторов, используемых в подпрограмме, должен быть описан только один раз.
* Каждый из идентификаторов, используемых в программе, должен быть описан один раз.
* Каждый из идентификаторов, используемых в блоке программе, должен быть описан один раз.
* Области видимости одинаковых идентификаторов не должны пересекаться.

**Условия соответствия видов значений:**

* Типы фактических и формальных параметров функций должны совпадать.
* Количество формальных и фактических параметров функции должно совпадать.

**Условия, определяющие количественные ограничения:**

* Количество формальных параметров функций не ограничено.
* Глубина вложенности блоков не ограничена.

# 5 Таблица соответствия языков

Соответствие языков программирования Kotlin и C# представлено в таблице 1.

Таблица 1 Соответствие языков Kotlin и C#

|  |  |
| --- | --- |
| Kotlin | C# |
| Структура программ | |
| Структура программ аналогична и представляет из себя набор выражений. | |
| Переменные | |
| Переменные объявляются с помощью ключевого слова var без указания типа.  <объявление переменной>:= val <название переменной> | val <название переменной> = <значение переменной>  Примеры:   * val n = 5; * val n. | Переменные объявляются без ключевого слова с указанием типа.  <объявление переменной>:= <тип переменной> <название переменной> | <тип переменной> <название переменной> = <значение переменной>  Примеры:   * int n/auto n; * int n = 5/auto n = 5. |
| Константы | |
| Константы объявляются с помощью ключевого слова val без указания типа.  <объявление константы>:= val  <имя константы> = <значение константы>  Пример:   * val n = 5. | Константы объявляются с помощью ключевого слова const c указанием типа.  <объявление константы>:= const <имя константы> = <значение константы>  Пример:   * const int n = 5. |
| Условия | |
| Условия оформляются с помощью ключевых слов if и else с применением фигурных скобок или без них.  <условие>:= if (<условие>) {<блок>} | if (<условие>) {<блок>} else {<блок>}  Пример:   * if (a < 5) {   foo(a)  println(a)  } else  foo(a) | Условия оформляются с помощью ключевых слов if и else с применением фигурных скобок или без них.  <условие>:= if (<условие>) {<блок>} | if (<условие>) {<блок>} else {<блок>}  Пример:   * if (a < 5) {   Foo(a);  Console.WriteLine(a);  } else  Foo(a); |
| Циклы | |
| Цикл for | |
| Цикл for объявляется с помощью ключевого слова for, фигурных скобок для тела цикла и круглых скобок для условия. При этом для параметра цикла не нужно указывать тип данных.  <цикл for>:= for (<параметр цикла> in <коллекция>)  {<блок>} |  for (<параметр цикла> in  <начальное значение> ..  <конечное значение>)  {<блок>}  Примеры:   * for (item in items) {   println(item)  }   * for (i in 0..9) {   println(i)  } | Цикл for объявляется с помощью ключевого слова for, фигурных скобок для тела цикла и круглых скобок для условия. При этом для параметра цикла необходимо указывать тип данных.  <цикл for>:= for (<тип параметра> <параметр цикла> in <коллекция>) {<блок>} | for (<тип параметра> <параметр цикла> = <начальное значение>); <параметр цикла> < <конечное значение>; <параметр цикла>++) {<блок>}  Примеры:   * for (int item in items) {   Console.WriteLine(item);  }   * for (int i = 0; i < 9; i++){   System.out.println(i);  } |
| Цикл while | |
| Цикл while оформляется с помощью ключевого слова while, фигурных скобок для тела цикла и круглых скобок для его условия.  <цикл while>:= while (<условие цикла>) {<блок>}  Пример:   * while (i > 0)   {  println(i)  i--  } | Цикл while оформляется с помощью ключевого слова while, фигурных скобок для тела цикла и круглых скобок для его условия.  <цикл while>:= while (<условие цикла>) {<блок>}  Пример:   * while (i > 0)   {  Console.WriteLine(i);  i--;  } |
| Функции | |
| Функции объявляются с помощью ключевого слова fun, фигурных скобок для оформления тела цикла, а также круглых скобок для оформления параметров  функции.  <объявление функции>:= fun <название функции>  (<параметры функции> )  {<блок>}  Пример:   * fun sum(a, b) {   return a + b  } | Функции объявляются с помощью типа возвращаемого значения, фигурных скобок для оформления тела цикла, а также круглых скобок для оформления параметров функции.  <объявление функции>:= <тип возвращаемого значения><название функции> (<параметры функции> ) {<блок>}  Пример:   * int sum(a, b) {   return a+b;  } |
| Процедуры | |
| Процедуры объявляются с помощью ключевого слова fun, фигурных скобок для оформления тела цикла, а также круглых скобок для оформления параметров процедуры.  Пример:   * fun sum(a, b) {   println(a+b)  } | Процедуры объявляются с помощью ключевого слова void, фигурных скобок для оформления тела цикла, а также круглых скобок для оформления параметров процедуры.  Пример:   * void sum(a, b) {   Console.WriteLine(a+b);  } |
| Работа с массивами | |
| Создание массива | |
| Массив создается с помощью функции arrayOf, в которую передаются элементы массива.  <объявление массива>:= val <название массива> = arrayOf (<элементы массива>)  Пример:   * val simpleArray =   arrayOf(1, 2, 3) | Массив создается с указанием типа элементов, которые в нем хранятся и фигурных скобок.  <объявление массива>:= <тип элементов> [] <название массива> := {<элементы массива>}  Пример:   * int[3] simpleArray =   {1, 2, 3} |
| Обращение к массиву | |
| Обращение к массиву создается с помощью квадратных скобок с номером элемента.  <обращение к массиву>:= <название массива> [<номер элемента>] | |
| Работа со списками | |
| Создание списка | |
| Список создается с помощью функции listOf.  <объявление списка>:= val <название списка> = listOf (<элементы массива>)  Пример:   * val simpleList =   listOf(1, 2, 3) | Список создается с помощью конструктора List.  <объявление списка>:= List <тип элементов> <название списка> = new List <тип элементов> ();  Пример:   * List<int> simpleList = new List<int>() {1, 2, 3}; |
| Обращение к списку | |
| Обращение к листу реализуется с помощью квадратных скобок.  <обращение к листу>:= <название списка>[<индекс элемента>] | |
| Добавление в список | |
| Добавление в лист осуществляется с помощью метода add.  <список>.add(<элемент>)  Пример:  simpleArray.add(5) | Добавление в лист осуществляется с помощью метода Add.  <список>.Add(<элемент>)  Пример:  simpleArray.Add(5) |
| Удаление из листа | |
| Удаление из листа осуществляется с помощью метода removeAt().  <список>.removeAt(<позиция>)  Пример:  simpleArray.removeAt(5) | Удаление из листа осуществляется с помощью метода RemoveAt().  <список>.RemoveAt(<позиция>)  Пример:  simpleArray.RemoveAt(5) |

# 6 Проект лексического анализатора

Лексер – часть транслятора, распознающая токены. Лексер преобразует последовательность символов программы в последовательность токенов.

Лексический анализ – это преобразование текста в значимые лексические лексемы, принадлежащие к категориям, определенным программой.

## 6.1 Модель данных

Для создания лексера был использован класс Lexer.

Для лексера целесообразно использовать отдельный класс по нескольким причинам. Использование отдельного класса для лексера повышает структурированность и модульность кода, что в итоге делает разработку, поддержку и тестирование программы более удобными и эффективными.

Для лексемы был создан класс Lexem. Для токена был создан класс Token. Использование отдельных классов было необходимо для корректной работы.

Диаграмма классов лексера представлена на рисунке 3.

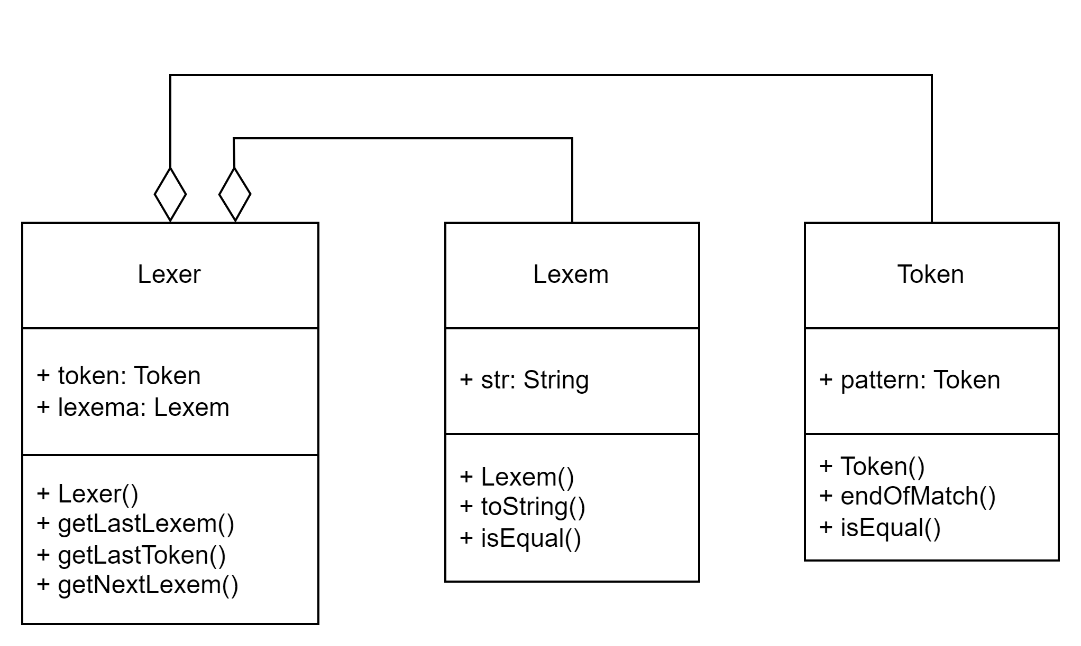


Рисунок 3 – Диаграмма классов лексера

## 6.2 Таблица ключевых слов

Ключевые слова обладают особым семантическим значением в контексте языка программирования. Они представляют собой зарезервированные слова, которые используются для определения ключевых конструкций языка, таких как условные операторы, циклы, определение функций.

Ключевые слова представлены в таблице 2.

Таблица 2 – Ключевые слова

|  |  |
| --- | --- |
| Ключевые слова | Ключ |
| var | VAR |
| val | VAL |
| fun | FUNCTION |
| while | WHILE |
| for | FOR |
| if | IF |
| when | WHEN |
| case | CASE |
| else | ELSE |
| break | BREAK |
| Char | CHARTYPE |
| String | STRING |
| Bool | BOOLEAN |
| Byte | BYTE |
| Short | SHORT |
| Int | INT |
| Long | LONG |
| is | IS |

## 6.3 Таблица зарезервированных имен

Зарезервированные имена не являются ключевыми словами или лексемами.

Зарезервированные имена являются именами, которые зарезервированы в языке программирования и не могут использоваться для определения пользовательских переменных, функций или классов. Они могут быть зарезервированы для специальных целей, таких как ключевые слова или стандартные библиотечные функции.

Отличие зарезервированных имен от лексем заключается в том, что лексемы – это отдельные элементы языка программирования, такие как идентификаторы, операторы, константы и знаки препинания, в то время как зарезервированные имена являются специальными словами, которые имеют определенное значение для компилятора или интерпретатора.

При создании лексера, программист должен учитывать зарезервированные имена и их использование в языке программирования, чтобы исключить их из возможных лексем и обеспечить корректный анализ исходного кода.

Зарезервированные имена представлены в таблице 3.

Таблица 3 – Зарезервированные имена

|  |  |
| --- | --- |
| Зарезервированное имя | Значение |
| listOf | LISTOF |
| arrayOf | ARRAYOF |
| readLine | READLINE |
| print | PRINT |
| println | PRINTLN |
| abs | ABS |
| ceil | CEIL |
| floor | FLOOR |
| max | MAX |
| min | MIN |
| round | ROUND |
| sqrt | SQRT |
| cbrt | CBRT |
| exp | EXP |
| log | LOG |
| pow | POW |
| sign | SIGN |

## 6.4 Типы лексем

Тип лексемы – это категория, к которой относится определенный символьный или строковый элемент исходного программного кода. В контексте лексического анализа типы лексем определяются для различных элементов языка программирования, например идентификаторы, ключевые слова, операторы, числовые и строковые литералы.

Типы лексем играют важную роль в обработке и анализе исходного кода, обеспечивая корректное выделение и классификацию лексем для последующего анализа и выполнения программы.

Типы лексем представлены в таблице 4.

Таблица 4 – Типы лексем

|  |  |
| --- | --- |
| Идентификатор | 1. Ключевое слово;  2. Зарезервированное имя;  3. var (VAR); |
| Оператор | 1. \* (MUL);  2. / ( DIV);  3. – (MINUS);  4. + (PLUS);  5. % (MOD);  6. = (EQUALS); |
| Комментарии | 1. Многострочный (RAWSTRING)  2. Обычный (COMMENT) |
| Разделители | 1. , (COMMA);  2. ( (LBR);  3. ) (RBR);  4. { (LCBR);  5. } (RCBR);  6. [ (LSQBR);  7. ] (RSQBR);  8. : (COLON); |
| Логические операторы | 1. ! (NOT);  2. && (AND);  3. || (OR);  4. == (ISEQUAL);  5. !== (ISNOTEQUAL);  6. >= (MOREOREQUAL);  7. <= (LESSOREQUAL);  8. > (MORE);  9. < (LESS); |
| Константы | 1. Константа (VAL); |
| Числа | 1. Число (NUM); |
| Строки | 1. Строка (STR); |
| Символы | 1. Символ (CHAR); |

## 6.5 Типы ошибок

Лексический анализатор выдает ошибку, если в ходе анализа ему на вход поступил неизвестный символ.

Например, на вход подаются следующий код на Kotlin:

var x = @

var y = x + 1

var z

z = 9.5

prin(false, 't')

Тогда лексический анализатор выдаст следующее сообщение об ошибке: «Произошла ошибка в ходе работы лексера. Неизвестный символ возле = ...»

## 6.6 Конечный автомат лексического анализатора

Конечный автомат в трансляции языков программирования – это математическая модель, которая используется для описания процесса анализа лексической структуры языка программирования. Конечный автомат представляет собой устройство с ограниченным набором состояний, где на вход поступает последовательность символов, и в зависимости от текущего состояния и входного символа автомат переходит в новое состояние.

Конечный автомат может использоваться для построения модели лексического анализатора. Его построение позволяет уменьшить объем кода и повысить производительность, обеспечить точность и эффективность работы программы, удобство разработки лексического анализатора, позволяет наглядно отобразить процесс лексического анализа.

Конечный автомат лексического анализатора представлен на рисунке 4.

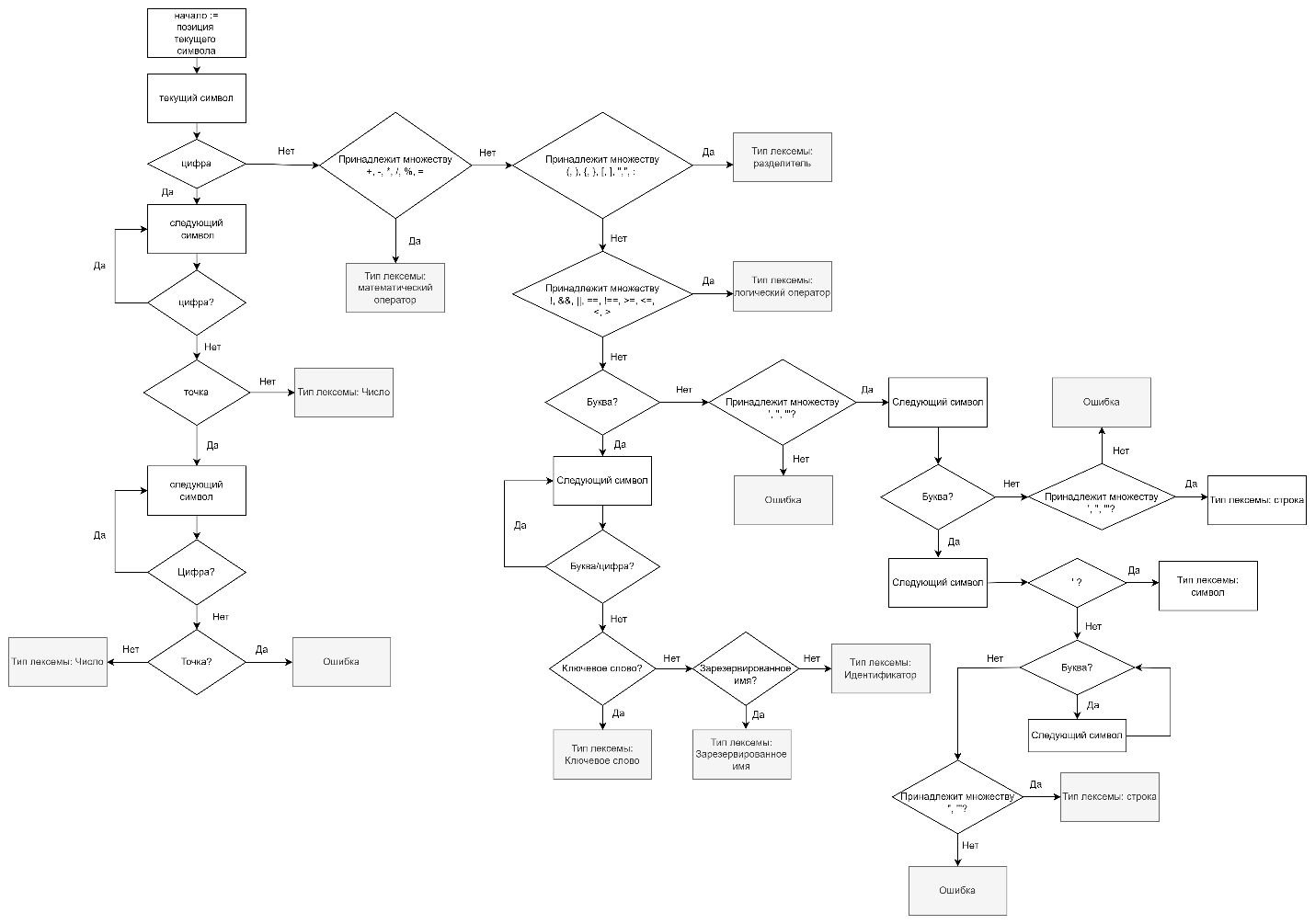


Рисунок 4 – Конечный автомат

Ниже представлен пример лексического анализа.

Входные данные:

fun prin(y: Boolean, x: Char): String {

return x

}

var y = 1

var z

z = 9.5

prin(false, 't')

Результат:

fun: FUNCTION

prin: ID

(: LBR

y: ID

:: COLON

Boolean: BOOL

,: COMMA

x: ID

:: COLON

Char: CHARTYPE

): RBR

:: COLON

String: STRING

{: LCBR

®: EOL

®: EOL

return: RETURN

x: ID

®: EOL

}: RCBR

®: EOL

®: EOL

var: VAR

y: ID

=: EQUALS

1: NUM

®: EOL

var: VAR

z: ID

®: EOL

z: ID

=: EQUALS

9.5: NUM

®: EOL

prin: ID

(: LBR

false: FALSE

,: COMMA

't': CHAR

): RBR

®: EOL

: EOF

# 7 Проект семантического анализатора

Семантический анализ — процесс сопоставления линейной последовательности лексем формального языка с его формальной грамматикой.

Для реализации семантического анализатора выбран метод рекурсивного спуска.

## 7.1 Модель данных

Для реализации семантического анализатора выбран язык Java. Для создания семантического анализатора был использован класс SemanticAnalyzer.

Для описания структуры данных семантического анализатора была сделана диаграмма моделей данных. Диаграмма моделей данных представлена на рисунке 5.

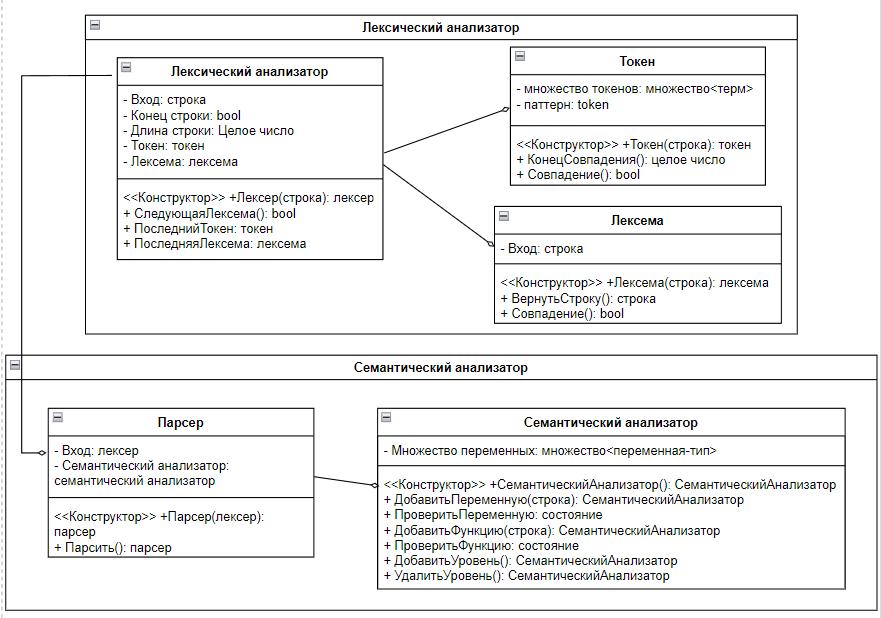


Рисунок 5 – Диаграмма моделей данных

## 7.2 Описание алгоритма рекурсивного спуска

Метод рекурсивного спуска — алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур, где каждая процедура соответствует одному из правил контекстно-свободной грамматики или БНФ.

Пусть G без левой рекурсии.

E→ T | T+E | T−E

T→ F | F∗T | F/T

F→ num | id | (E)

## 7.3 Пример дерева разбора

Входные данные:

var x = 2

Результат семантического анализа:

program

statement

declaration

simple-declaration

VAR

x

EQUALS

term

value

NUM

EOL

EOF

## 7.4 Типы ошибок

1. Входные данные: пустой файл

Выходные данные:

Файл пуст!

Произошла ошибка в ходе работы пасрера…

2. Входные данные:

2

Выходные данные:

Ошибка! Ожидалось выражение!

3. Входные данные:

var x=2 var y=1

Выходные данные:

Ошибка! Ожидался конец строки!

4. Входные данные:

var =2

Выходные данные:

Ошибка! Ожидался ID!

Ошибка! Ожидалось объявление!

Ошибка! Ожидалось выражение!

5. Входные данные:

var = x = 1 +

Выходные данные:

Ошибка! Ожидалась правая часть!

6. Входные данные:

var x = 7\*(1 +2

Выходные данные:

Ошибка! Ожидалась скобка!

7. Входные данные:

fun prin(x: Char): String {

return x

}

prin(1)

Выходные данные:

Ошибка! Функция вызвана не с теми параметрами!

8. Входные данные:

fun prin(x: Char): String {

return x

}

prin

Выходные данные:

Ошибка! Ожидался вызов функции!

9. Входные данные:

val y

Выходные данные:

Ошибка! Необходимо инициализировать константу

10. Входные данные:

If ()

Выходные данные:

Ошибка! Ожидалось логическое выражение!

11. Входные данные:

while ()

Выходные данные:

Ошибка! Ожидалось условие цикла!

Ошибка! Ожидался цикл!

12. Входные данные:

fun

Выходные данные:

Ошибка! Ожидалось название функции!

Ошибка! Ожидалось объявление функции!

13. Входные данные:

for()

Выходные данные:

Ошибка! Ожидался идентификатор!

Ошибка! Ожидалось выражение!

# 8 Проект синтаксическогоо анализатора

Семантический анализ — этап в последовательности действий алгоритма обработки текста формального языка, заключающийся в выделении семантических отношений, формировании семантического представления текста формального языка.

## 8.1 Модель данных

Для реализации семантического анализатора выбран язык Java. Для создания семантического анализатора был использован класс SemanticAnalyzer. Для описания структуры данных семантико-синтаксического анализатора, включающего семантический анализатор, была сделана диаграмма моделей данных. Диаграмма моделей данных представлена на рисунке 5 (глава 7.1).

## 8.2 Методы проверки нарушений контекстных условия

Семантический анализ сверяет дерево синтаксического анализа с таблицей символов и определяет, является ли оно семантически непротиворечивым. Этот процесс включает проверку типов данных, проверку меток и проверку управления потоком.

## 8.3 Типы ошибок

1. Входные данные:

var x = a + 1

Выходные данные:

Переменная a не объявлена перед использованием

2. Входные данные:

fun prin(x: Char): Char{

return x

}

prin(1)

Выходные данные:

Ошибка! Функция вызвана не с теми параметрами!

# 9 Проект генератора кода

Генерация кода — процесс конвертации исходной программы в эквивалентную целевую программу.

## 9.1 Модель данных

Для реализации генератора кода выбран язык Java. Для создания генератора кода был использован класс CodeGenerator. Для описания структуры данных генератора кода была сделана диаграмма моделей данных. Диаграмма моделей данных, включающая в себя генератор кода, представлена на рисунке 6.

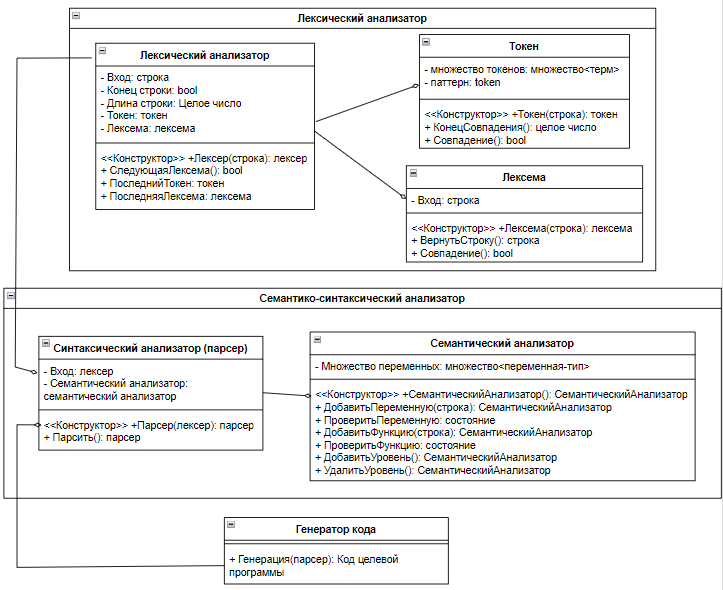


Рисунок 6 – Диаграмма моделей данных с генератором кода

## 9.2 Метод кодогенерации

На вход генератор получает промежуточное представление исходной программы от начальной стадии компилятора и выводит эквивалентную целевую программу.

## 9.3 Примеры кодогенерации

1. Входные данные:

var x = 5

val y = x + 7

Выходные данные:

using System;

class Trans {var x = 5;

var y = x+7;

}

**10 Проект оптимизатора кода**

Модуль оптимизации кода при трансляции с одного языка на другой в программировании представляет собой инструмент, который осуществляет оптимизацию исходного кода программы перед его генерацией на другой язык программирования. Этот модуль выполняет различные оптимизации, такие как удаление неиспользуемого кода, улучшение производительности, оптимизацию алгоритмов и структур данных, а также улучшение читаемости и поддерживаемости кода.

**10.1 Модель данных**

Модель данных оптимизатора кода представлена на рисунке 7.

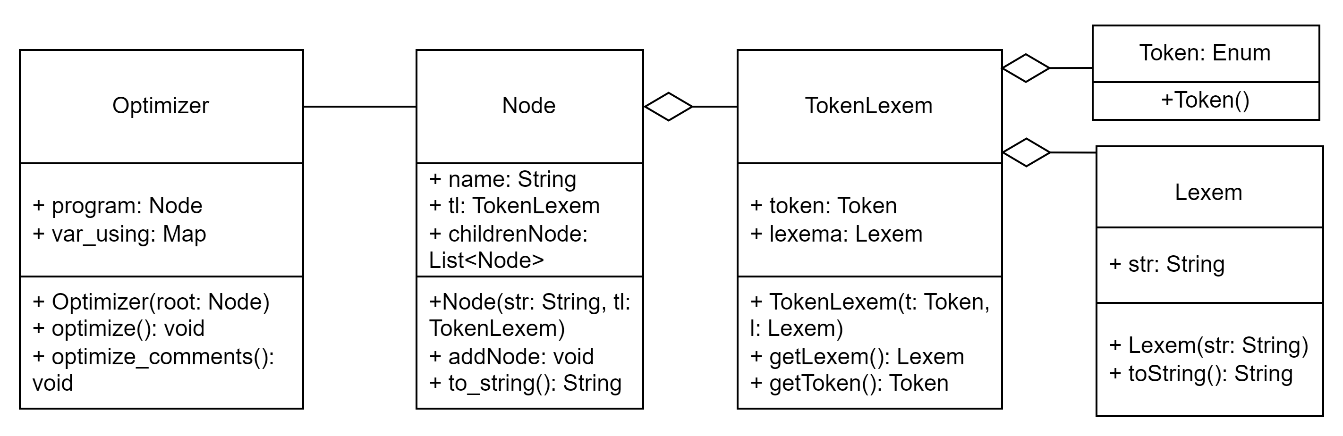


Рисунок 7 – Модель данных оптимизатора кода

**10.2 Методы оптимизации кода**

В качестве методов оптимизации кода в данном случае были использованы методы удаления неиспользуемых переменных и констант, а комментариев.

**10.3 Примеры оптимизации кода**

Входные данные:

var x = 2

var z = 6

/\* Какой-то коммент

Многострочный \*/

var y = 7

if (1 < x) {

x = x + 1

var nfnf = 6

}

Выходные данные:

using System;

class Trans {var x = 2;

if (1 < x) {

x = x+1;

}

}

# 11 Тестирование

## 11.1 Лексический анализатор

Результаты тестирования лексического анализатора представлены в таблице 5.

Таблица 5 – Тестирование лексического анализатора

|  |  |  |
| --- | --- | --- |
| Тест-кейс | Входные данные | Выходные данные |
| Пустой файл |  |  |
| Ключевые слова |  |  |
| Зарезервированные слова |  |  |
| Идентификаторы |  |  |
| Объявление переменных |  |  |
| Объявление констант |  |  |
| Числа |  |  |
| Переходы на новую строку |  |  |
| Строки |  |  |
| Символы |  |  |
| Разделители |  |  |
| Математические операторы |  |  |
| Логические операторы |  |  |
| Комментарии |  |  |
| Неожиданные символы |  |  |

## 11.2 Синтаксический анализатор

## 11.3 Семантический анализатор

# 11.4 Оптимизатор кода

# 11.5 Генератор кода

someFn()

someFn(a, b)

someFn(a)

someFn(1+3)

someFn(x\*1)

someFn(a || b)

someFn(!a)

someFn(!a, a||b, c&&d)

someFn(!a, -a)

a=b

a = b&&c

a = b && c ||d

a = -d

a = b+7

a = (x/a)\*1 - 1

a = !b

a = (a || b)

if (a<b){

a=b

b=a

func()} else{}

if (a<b){}

if (a<b){} else{}

var p = myVar && myVar

var p = myVar || myVar

var p = myVar && !myVar

var p = !myVar && myVar

var x = !!b

var x = a && !!b

var x = a && !(a > 5)

var x = !!!b

var y = (a < 7) && b

var y = (a && b)

var y = !true && false

var y = true || (false && false) && my

var y = true || false || my

var y = true || false || !my

var y = (5 - 4) > 1

when(a) {

1 -> {}}

when(a) {1 -> {}

2 -> {}

}

when(a) {1 -> {}

2 -> {a=b}

}

when(a) {1 -> {}

2 -> {a=b}

3 -> {

a = a+b

}

}

for(a in b) {}

a = 1+2

a = 1+2-3-6-4-2

a = (b+c - 0)

a = 1+(2+3)

for(a in 1..2){

if (a < b) {a = a+b}

a = 7+1

a = 3+1+2

}

for(a in b){

if (a < b) {a = a+b}

a = 7+1

a = 3+1+2

continue

break

}

for(a in 1..2){

if (a < b) {

a = a+b

}

continue

if (a < b) {

a = a+b

}

}

public int double (int x) {

return 2\*x;

}

if (a < b) {

a=b

} else {}

if (a < b) { a=b}

a=b

var a = 1

b = a ||b

b = a&&(!b || (c && !d))

b = 4.3

while (a <99) {

a = a+1

}

while (a <99)

{

a = a+1

}

return x+44

return x

return (a || b)

# Заключение

Даже эмбрион знает, как писать заключение :)

Список литературы

<https://kotlinlang.org/docs/basic-types.html>

https://translated.turbopages.org/proxy\_u/en-ru.ru.1629bc8b-64525281-b29c5175-74722d776562/https/kotlinlang.org/api/latest/jvm/stdlib/kotlin/