МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Дальневосточный федеральный университет»**

**ШКОЛА ЕСТЕСТВЕННЫХ НАУК**

**Кафедра прикладной математики, механики, управления и программного обеспечения**

РАЗРАБОТКА ТРАНСЛЯТОРА, ПЕРЕВОДЯЩЕГО ПОДМНОЖЕСТВО ЯЗЫКА JAVA В ЭКВИВАЛЕНТНОЕ ПОДМНОЖЕСТВО ЯЗЫКА RUBY

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Теория языков программирования и компиляторы»

по образовательной программе подготовки бакалавров по направлению 09.03.04 Программная инженерия

Выполнил студент гр. Б8118-09.03.04

\_\_\_\_\_\_\_\_\_\_\_ Туров С.Е.

Руководитель: доцент кафедры ПММУиПО Лемза А.О.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

(подпись)

Защищён с оценкой \_\_\_\_\_\_\_\_

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

(подпись) И.О. Фамилия

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2021 г.

г. Владивосток

2021 г.

# Оглавление

[Введение 3](#_Toc75420268)

[1 Неформальная постановка задачи 4](#_Toc75420269)

[2 Синтаксис входного языка 5](#_Toc75420270)

[3 Контекстные условия 10](#_Toc75420271)

[4 Таблица соответствия 12](#_Toc75420272)

[5 Проект лексического анализатора 16](#_Toc75420273)

[5.1 Таблица ключевых слов языка Java 16](#_Toc75420274)

[5.2 Типы лексем 17](#_Toc75420275)

[5.3 Ошибки лексического анализатора 19](#_Toc75420276)

[5.4 Конечный автомат лексического анализатора 19](#_Toc75420277)

[6 Проект синтаксического анализатора 23](#_Toc75420278)

[Модель данных 23](#_Toc75420279)

[Проект алгоритма parse 29](#_Toc75420280)

[Синтаксический анализатор 29](#_Toc75420281)

[Пример дерева разбора для следующей входной цепочки 30](#_Toc75420282)

[Ошибки синтаксического анализатора 32](#_Toc75420283)

[Заключение 33](#_Toc75420284)

[Список литературы 34](#_Toc75420285)

# Введение

В современном мире компиляторы играют большую роль в программировании. Причём, это не только компиляторы языков высокого уровня в языки более низкого уровня, но и компиляторы из одного высокоуровневого языка в другой.

Компилятор – программа или техническое средство, выполняющее компиляцию.

Компиляция – трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком к машинному коду. Входной информацией для компилятора является описание алгоритма или программа на проблемно-ориентированном языке, а на выходе компилятора – эквивалентное описание алгоритма на машинно-ориентированном языке.

Трансляция – преобразование программы, представленной на оном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т.д. [1].

**Целью курсового проекта** является разработка проекта транслятора, который преобразует программу, содержащую подмножество языка Java, в программу, содержащую подмножество языка Ruby, генерируя эквивалентный исходный код. Язык задается множеством цепочек, принадлежащих ему.

**Задачи курсового проекта**:

* разработать грамматику подмножества языка Java;
* описать контекстные условия входного языка;
* описать соответствие конструкций входного и выходного языков;
* разработать проект лексического анализатора;
* разработать проект синтаксического анализатора.

# 1 Неформальная постановка задачи

Неформальной постановкой задачи курсового проекта является разработка проекта транслятора из подмножества языка **Java** в эквивалентное подмножество языка **Ruby**.

Подмножество языка Java включает:

1. типы данных: byte, short, int, long, float, double, char, многомерные аналоги перечисленных типов;
2. операторы:
3. операторы присваивания: =, +=, -=, \*=, /=;
4. унарные арифметические операторы: ++, --;
5. бинарные арифметические операторы: +, -, \*, /, %;
6. операторы отношения: >, <, >=, <=, !=, ==;
7. логические операторы: ||, &&
8. побитовые бинарные операторы: |, &;
9. операторы циклов:
10. do … while,
11. while,
12. for;
13. операторы ветвления:
14. if,
15. if …else;
16. стандартные функции:
17. функция вывода в консоль System.out.println();
18. операторы блоков инструкций:
    1. <тип> <название> ( <набор входных данных> ) { <набор инструкций> }
19. операторы возврата из подпрограммы:
20. break
21. return <число / переменная>

# 2 Синтаксис входного языка

На вход в компилятор поступает цепочка, принадлежащая языку Java. Для корректной обработки входных данных необходимо разобрать синтаксис входного языка Java.

G = (N, ∑, P, <программа>);

N = {<программа>, <имя текущего пакета>, <подключение пакетов>, <подключение пакета>, <имя пакета>, <объявление класса>, <тело класса>, <объявление переменной>, <объявление константы>, <значение>, <объявление функции>, <вывод в консоль>, <содержимое вывода>, <параметры функции>, <тип данных>, <тип данных функции>, <символ>, <цифра>, <целое число>, <вещественное число>, <число>, <строка>, <идентификатор>, <символ идентификатора>, <ид>, <тело функции>, <возврат значения>, <блок кода>, <цикл>, <ветвление>, <арифметический оператор>, <унарный арифметический оператор> , <символьное значение> , <строковое значение>, <оператор сравнения>, <унарная арифметическая операция>, <арифметический операнд>, <арифметическое выражение> , <операнд сравнения>, <выражение сравнения>, <логический бинарный оператор>, <логическое значение>, <логический операнд>, <логическое выражение>, <побитовый бинарный оператор>, <побитовый операнд>, <побитовое выражение>, <инструкция>, <выражение>, <вызов функции>, <параметры вызова функции>, <оператор присваивания>, <присваивание>, <имя функции>, <имя переменной>, <имя константы>, <тип данных переменной>, <главная функция>, <прочие символы>, <любая строка>, <тело цикла>, <оператор цикла>, <знак числа>, <имя или инициализация>, <следующая константа>}

∑ = {**a, b, c, …, x, y, z, A, B, C, … X, Y, Z, а … я, А … Я, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (, ), {, }, [, ], +, -, \*, /, %, >, <, =, !, &, |, ; , “ , ‘, , , \_, # , @, $, ^, ~, №, :, ?**};

P: {

<программа> → **package** <имя текущего пакета> <подключение пакетов> | **package** <имя текущего пакета> <объявление класса>

<подключение пакетов> → <подключение пакета> <подключение пакетов> | <объявление класса>

<объявление класса> → **class Main {**<главная функция> <тело класса>**}** | **class Main {**<главная функция>}

<тело класса> → <объявление переменной> <тело класса> | <объявление переменной> | <объявление функции> <тело класса> | <объявление функции> | <объявление константы> <тело класса> | <объявление константы>

<подключение пакета> → **import** <имя пакета>

<имя пакета> → **java.lang.Math**

<объявление переменной> → <тип данных переменной> <имя или инициализация>

<имя или инициализация> → <имя переменной>**,** <имя или инициализация> | <имя переменной> **=** <значение>**,** <имя или инициализация> | <имя переменной>**;** | <имя переменной> **=** <значение>**;**

<объявление константы> → **const** <тип данных переменной> <следующая константа>

<следующая константа> → <имя переменной> **=** <значение>, <следующая константа> | <имя переменной> **=** <значение>**;**

<значение> → <число> | <строковое значение> | <символьное значение> | <логическое значение>

<объявление функции> → <тип данных функции> <имя функции> **(**<параметры функции>**)** **{**<тело функции>**}**

<параметры функции> → <тип данных переменной> <идентификатор>| <тип данных переменной> <идентификатор>**,** <параметры функции>

<тип данных> → **boolean | byte | short | char | int | float | double**

<тип данных функции> → <тип данных> | **void**

<тип данных переменной> → <тип данных>

<символ> → **a |…| z | A |… | Z**

<цифра>→ **0|1|2|3|4|5|6|7|8|9**

<целое число> → <цифра> <целое число> | <цифра>

<вещественное число> → <целое число>.<целое число>

<число> → <целое число> | <вещественное число>

<строка> → <символ><строка>|<цифра><строка>| <символ> | <цифра> |\_<строка>

<прочие символы> → **а| … | я | А | … | Я | \_ | ! | @ | # | $ | % | ^ | & | \* | ( | ) | - | + | [ | ] | { | } | ~ | № | ; | : | ? | = | < | >| . | ,**

<любая строка> → <символ> <любая строка> | <цифра> <любая строка> | <прочие символы> <любая строка> | <символ> | <цифра> | <прочие символы>

<символ идентификатора> → <символ>

<идентификатор> → <символ идентификатора> <ид> | <символ идентификатора>

<ид> → <символ идентификатора> <ид> | <цифра> <ид> | <символ идентификатора> | <цифра>

<тело функции> → <блок кода> <возврат значения> | <блок кода>

<возврат значения> → **return** <выражение>**;**| **return** <имя переменной>; | **return** <имя константы>;

<блок кода> → <инструкция> <блок кода> | <инструкция>;

<цикл> → **while (**<логическое выражение>**)** **{**<тело цикла>**}** | **do {**<тело цикла>**} while (**<логическое выражение>**);** | **for** (<инструкция>**;** <логическое выражение>**;** <инструкция>**) {**<тело цикла>**}**

<тело цикла> → <блок кода> <оператор цикла> <блок кода> | <оператор цикла> <блок кода> | <блок кода> <оператор цикла> | <блок кода> | <оператор цикла>

<оператор цикла> → **break | continue**

<ветвление> → **if (**<логическое выражение>**) {**<блок кода>**} | if** **(**<логическое выражение>**) {**<блок кода>**} else {**<блок кода>**} | if (**<логическое выражение>**) {**<блок кода>**} else** <ветвление>

<арифметический оператор> → **+ | - | / | \* | %**

<унарный арифметический оператор> → **++ | --**

<знак числа> → **+ | -**

<символьное значение> → **’**<символ>**’** | **’**<цифра>**’** | **‘**<арифметический оператор>**’** | **‘**<прочие символы>**’**

<строковое значение> → **“**<любая строка>**”**

<оператор сравнения> → **== | != | > | < | >= | <=**

<унарная арифметическая операция> → <идентификатор> <унарный арифметический оператор> | <унарный арифметический оператор> <идентификатор> | <знак числа> <идентификатор> | <знак числа> <число>

<арифметический операнд> → <число> | <имя переменной> | <вызов функции> | **(**<арифметическое выражение>**)** | <унарная арифметическая операция>

<арифметическое выражение> → <арифметический операнд> <арифметический оператор> <арифметический операнд>

<операнд сравнения> → <число> | <символьное значение> | <имя переменной> | <вызов функции>

<выражение сравнения> → <операнд сравнения> <оператор сравнения> <операнд сравнения>

<логический бинарный оператор> → **&& | ||**

<логическое значение> → **true | false**

<логический операнд> → <логическое значение> | <выражение сравнения> | **(**<логическое выражение>**)** | <имя переменной> | <вызов функции>

<логическое выражение> → **!**<логический операнд> | <логический операнд> <логический бинарный оператор> <логический операнд> | <логический операнд>

<побитовый бинарный оператор> → **&** | **|**

<побитовый операнд> → <число> | <имя переменной> | <вызов функции> | (<побитовое выражение>)

<побитовое выражение> → <побитовый операнд> | <побитовый операнд> <побитовый бинарный оператор> <побитовый операнд>

<инструкция> → <присваивание>**;** | <объявление переменной> | <объявление константы> | <вызов функции>**;** | <выражение>**;** | <цикл> | <ветвление> | <вывод в консоль>

<вывод в консоль> → **System.out.println(** <содержимое вывода> **) ;**

<содержимое вывода> → <идентификатор>**,** <содержимое вывода> | <идентификатор>

<выражение> → <арифметическое выражение> | <логическое выражение> | <унарная арифметическая операция> | <значение> | <побитовое выражение>

<вызов функции> <имя функции> **(**<параметры вызова функции>**)**

<параметры вызова функции> → <выражение>| <выражение>, <параметры вызова функции> |

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

<присваивание> → <имя переменной> <оператор присваивания> <выражение>

<имя функции> → <идентификатор>

<имя переменной> → <идентификатор>

<имя константы> → <идентификатор>

<главная функция> → **public static void main (String[] args) {** <блок кода> **}**

}

# Контекстные условия

Определим контекстные условия [2] для языка программирования Java. Приведем следующую классификацию контекстных условий:

1. Контекстные условия о правилах объявления идентификаторов в программах.

* все используемые в программах идентификаторы должны быть объявлены непосредственно до их использования в теле программы;
* каждый из идентификаторов, используемых в программе, должен иметь уникальное имя и быть объявлен один раз с указанием принадлежности к типу данных в области своего действия [3].

1. Контекстные условия о правилах использования идентификаторов в своей области действия.

* каждому вхождению идентификатора в списке спецификаций должно соответствовать его вхождение в списке формальных параметров;
* идентификатор, входящий в список значений, должен входить в список спецификаций [3].

1. Контекстные условия, определяющие правила соответствия видов величин, входящих в синтаксические конструкции программ.

* при использовании стандартных пакетов языка Java, в функциях должны использоваться операнды, чьи типы данных совпадают с запросом использующей их функции и/или процедуры;
* операндами арифметических выражений могут быть только числовые константы, идентификаторы арифметических переменных, указатели арифметических функций;
* в арифметических операциях переменные должны быть одного типа или иметь возможность быть приведенными к общему типу;
* при переводе операторов процедур или указателей процедур-функций проверяется соответствие видов значений фактических и формальных параметров, а также их количественное равенство [3].

1. Контекстные условия, задающие ограничения реализации.

* в выражениях в теле программы должна сохраняться вложенность скобок (на каждую открывающуюся скобку должна приходиться закрывающая скобка);
* в языке Java каждое приложение должно содержать главную функцию public static void main [3];
* в конце каждой последовательности операций и/или вызовов функций, соответствующих синтаксису языка, должен быть установлен знак “;”.

# 4 Таблица соответствия

При помощи таблицы соответствия отображаются элементы языка Java, эквивалентные элементам языка Ruby. В ней отображаются как представление отдельных лексем, так и структура программы, процедур и функций.

Таблица 1 - Таблица соответствия языков Java и Ruby

|  |  |
| --- | --- |
| **Java** | **Ruby** |
| 1. Структура программы  import <имя пакета>  <объявление главного класса>  <объявление констант>  <объявление переменной>  <объявление функций>  <главная функция> | 1. Структура программы  <объявление констант>  <объявление переменной>  <объявление функций>  <главная функция> |
| 2. Объявление переменной  2.1. <тип данных переменной> <имя переменной> ;  2.2. <тип данных переменной> <имя переменной> = <значение переменной> ; | 2. Объявление переменной  2.1. <имя переменной>  2.2. <имя переменной> = <значение переменной> |
| 3.Циклы  3.1. while (<условие>) {<тело цикла>} | 3.Циклы  3.1. while <условие> do  <тело цикла>  end |

Продолжение таблицы 1

|  |  |
| --- | --- |
| 3.Циклы  3.2. do {<тело цикла>} while (<условие>)  3.3. for (<инициализация>; <условие>; <итерация>) {<блок>} | 3.Циклы  3.2. while <условие> do  <тело цикла>  end  3.3. for <инициализация> in <условие> do  <тело цикла>  end |
| 4. Условные переходы   |  | | --- | | 4.1. if (<условие>) {<тело>}  4.2. if (<условие>) {<тело>} else {<тело>}  4.3. if (<условие>) {<тело>} else <ветвление> | | 4. Условные переходы  4.1. if <условие>  <тело>  end  4.2. if <условие>  <тело>  else  <тело>  end  4.3. if <условие>  <тело>  else  <ветвление> |
| 5.Типы данных  boolean  byte  short  int  float  double  char  void | 5.Типы данных  bool  bytes  int  int  float  double  str  (отсутствует) |

Продолжение таблицы 1

|  |  |
| --- | --- |
| **Java** | **Ruby** |
| 6. Логические типы данных  true  false | 6. Логические типы данных  true  false |
| 7. Логические операции  !  ||  && | 7. Логические операции  !  ||  && |
| 8. Модификаторы типа данных  const | 8. Модификаторы типа данных  (отсутствует) |
| 9. Математические операции  +  -  \*  /  %  ++  -- | 9. Математические операции  +  -  \*  /  %  (отсутствует)  (отсутствует) |
| 10.Операторы сравнения  ==  !=  >  <  >=  <= | 10.Операторы сравнения  ==  !=  >  <  >=  <= |

Продолжение таблицы 1

|  |  |
| --- | --- |
| **Java** | **Ruby** |
| 11. Побитовые операторы  &  | | 11. Побитовые операторы  &  | |
| 12. Операторы присваивания  =  +=  -=  \*=  /=  %= | 12. Операторы присваивания  =  +=  -=  \*=  /=  %= |
| 13. Главная функция  public static int main (String[] args]) {  …  return 0;} | 13. Главная функция  def main()  … |
| 14. Функция  int fName (){  …  return <выражение>;  } | 14. Функция  def fName()  …  return <выражение>  end |
| 15. Процедура  void pName (){  …  } | 15. Процедура  def pName():  …  end |
| 16. Стандартная функция  System.out.println(); | 16. Стандартная функция  puts() |

# 5 Проект лексического анализатора

## 5.1 Таблица ключевых слов языка Java

Таблица 2 - Ключевые слова языка Java

|  |  |
| --- | --- |
| **Ключевые слова** | **Ключи** |
| break | K1 (оператор выхода из цикла) |
| const | K2 (модификатор типа данных) |
| continue | K3 (оператор продолжения цикла) |
| while | K4 (оператор цикла) |
| do | K5 (оператор цикла) |
| if | K6 (оператор ветвления) |
| else | K7 (оператор ветвления) |
| for | K8 (оператор цикла) |
| import | K9 (директива предпроцессора) |
| public | K10 (модификатор доступа) |
| private | K11 (модификатор доступа) |
| protected | K12 (модификатор доступа) |
| static | K13 (модификатор принадлежности) |
| Main | K14 (основной класс) |
| main | K15 (основная функция) |

Таблица 3 - Зарезервированные слова Java

|  |  |
| --- | --- |
| **Зарезервированное имя** | **Значение** |
| class | R1 (тип данных) |
| boolean | R2 (тип данных) |
| byte | R3 (тип данных) |
| short | R4 (тип данных) |
| int | R5 (тип данных) |
| float | R6 (тип данных) |
| double | R7 (тип данных) |
| char | R8 (тип данных) |
| void | R9 (тип данных) |
| String | R10 (тип данных) |
| true | R11 (логическое значение) |
| false | R12 (логическое значение) |

## 5.2 Типы лексем

Тип лексемы – это код, который говорит о том, что данная лексема принадлежит одной из обозначенных нами групп. К примеру, лексема может быть ключевым словом, тогда в поле типа будет стоять соответствующая буква (К). Номер лексемы уточняет конкретное значение этой лексемы. Например, если было ключевое слово if, то ключ лексемы будет содержать число, соответствующее if в таблице ключевых слов. Если тип лексемы — идентификатор (ключевое слово), то ключ лексемы будет значением идентификатора в таблице имен, которую создаёт лексический анализатор. Если тип лексемы — число, то значение лексемы будет ссылкой на таблицу со значением числа. Типы лексем представлены в таблице 4.

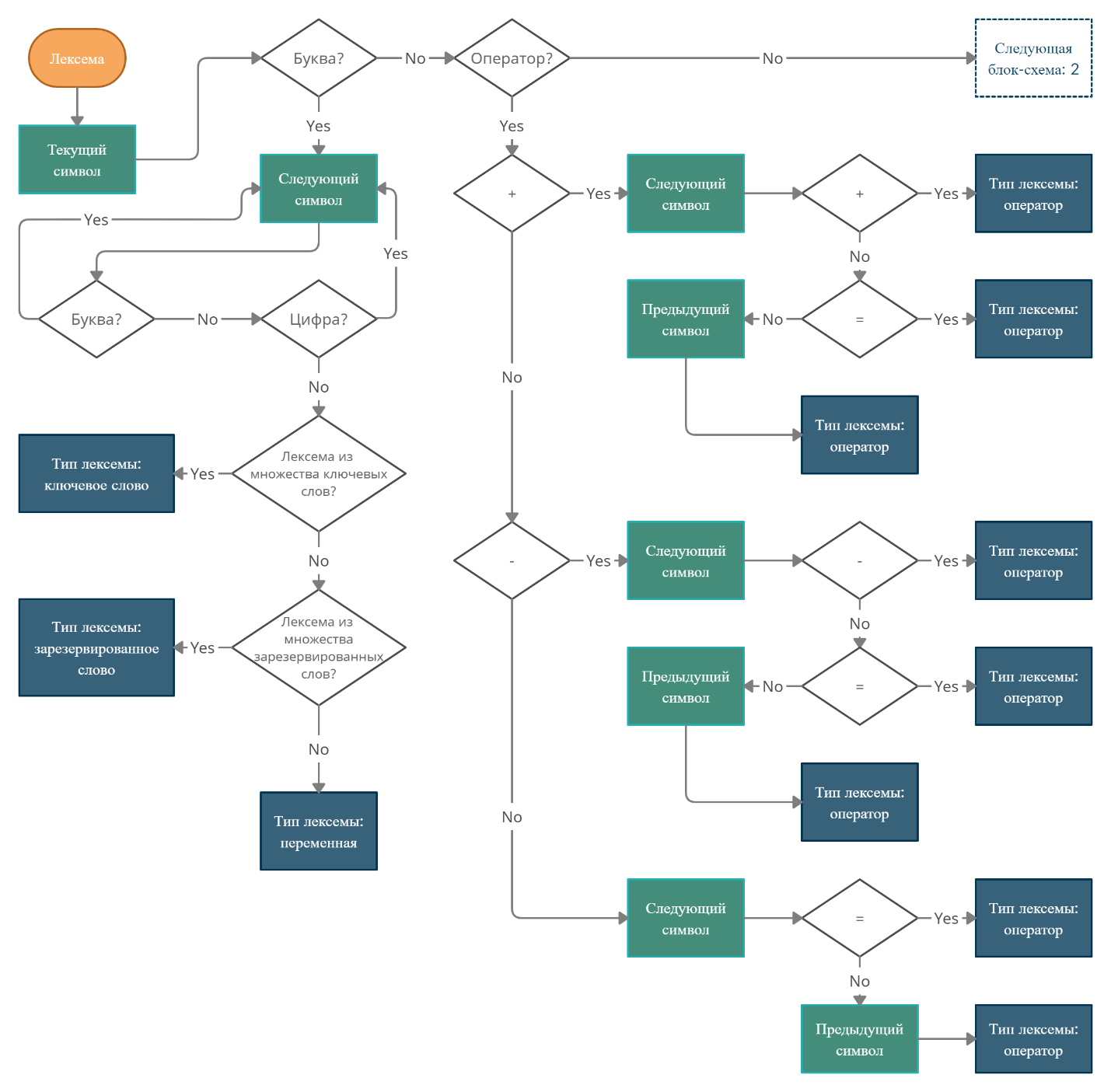
Таблица 4 – Таблица лексем

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Идентификатор | Ключевое слово  Зарезервированное слово  Переменная | | | K-  R-  I3 |
| Численный | Целое число  Вещественное число | | | NN |
| Символьный | Символ, заключенный в одинарные кавычки (символьный)  Последовательность символов, заключенная в двойные кавычки (строковый) | | | I3  I3 |
| Оператор | | +  ++  +=  -  --  -=  \*  \*=  /  /=  %  %=  |  ||  &  &&  !  !=  >  >=  <  <=  =  == | О1  О2  О3  О4  О5  О6  О7  О8  О9  О10  О11  О12  О13  О14  О15  О16  О17  О18  О19  О20  О21  О22  О23  О24 | |
| Разделители | | .  ,  ;  {  }  (  )  [  ] | D1  D2  D3  D4  D5  D6  D7  D8  D9 | |
| Строковые разделители | | ‘  “ | Q1  Q2 | |

## 5.3 Ошибки лексического анализатора

1. Некорректно задана лексема численного типа (после последней цифры числа обнаружена лишняя точка, либо строковый разделитель, либо буква).
2. Некорректно задана лексема строкового типа (после строкового разделителя встречена цифра, буква или ещё один строковый разделитель).
3. Некорректно задана лексема символьного типа (после строкового разделителя встречена цифра, буква или ещё один строковый разделитель).
4. Не встречена закрывающая кавычка во время чтения лексемы строкового типа (символов между разделителями больше одного, либо отсутствует закрывающий строковый разделитель (одинарная кавычка)).
5. Не встречена закрывающая кавычка во время чтения лексемы строкового типа (не обнаружен закрывающий строковый разделитель (двойная кавычка)).
6. Некорректно заданная лексема (не нашлось такой точки входа в лексическом анализаторе, которая распознаёт текущий символ входного текста).

## 5.4 Конечный автомат лексического анализатора

  
Рисунок 1 – Первая часть блок-схемы конечного автомата лексического анализатора

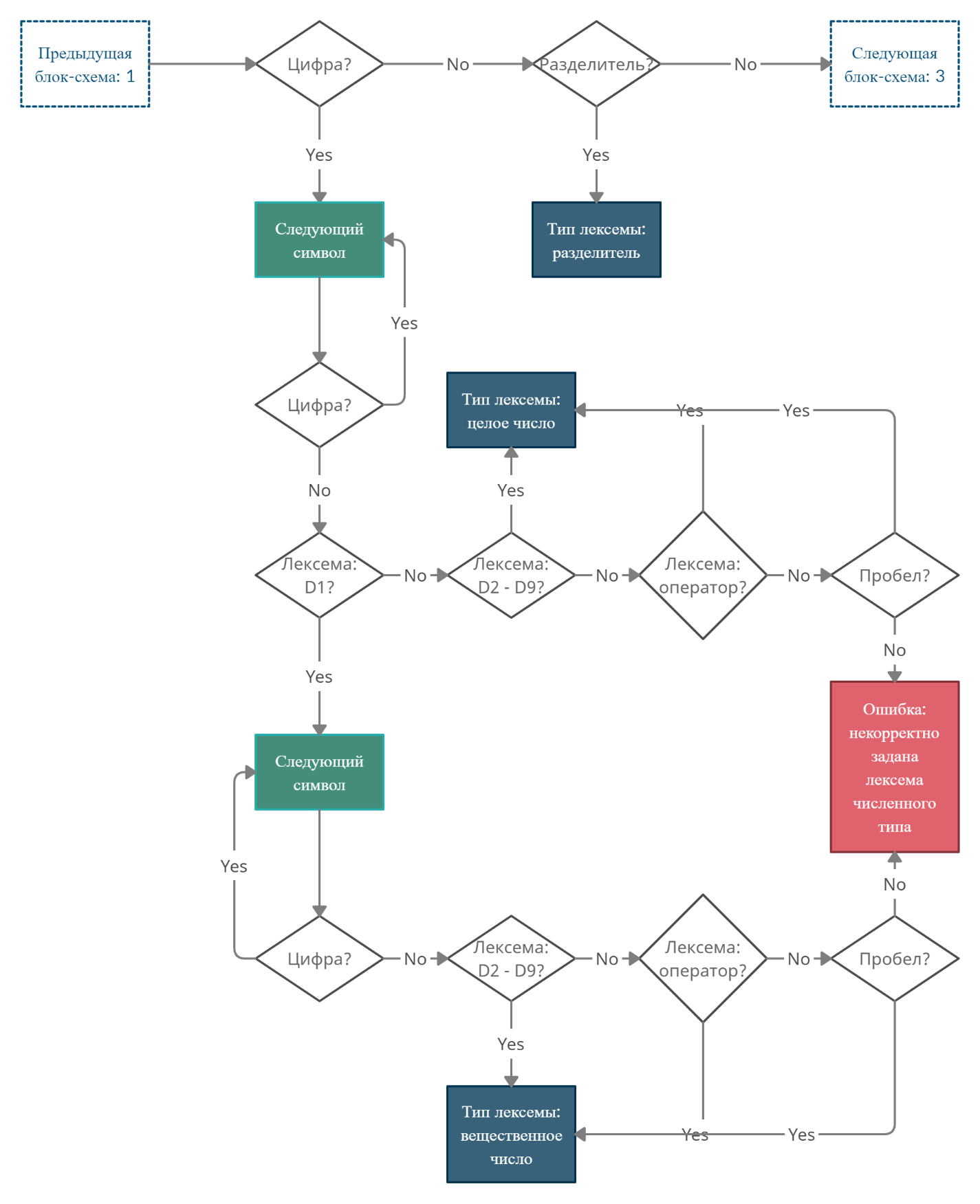
  
Рисунок 2 – Вторая часть блок-схемы конечного автомата лексического анализатора

  
Рисунок 3 – Третья часть блок-схемы конечного автомата лексического анализатора

# 6 Проект синтаксического анализатора

Задача синтаксического анализа – проверка правильности структуры программы, описываемой контекстно-свободной грамматикой. Класс контекстно-свободных грамматик шире класса автоматных грамматик, поэтому и синтаксический анализатор сложнее лексического.

Для синтаксического анализатора был выбран алгоритм построения правого разбора по списку разбора, получаемого с помощью алгоритма Эрли, оперирующий табличным списком разбора.

# Модель данных

Язык реализации – C++. Алгоритм разбора реализован в классе Parser. Для работы необходимо проинициализировать объект этого класса: задать ему формальную грамматику для разбора класса Grammar и лексический анализатор класса Lexical\_Analyzer. Метод parse класса Parser получает в качестве параметра входную цепочку, содержащую текст исходной программы, написанный на Java, и возвращает массив, содержащий правый разбор входной цепочки в случае успеха, либо код ошибки синтаксического анализатора в случае неудачи.

Формальная грамматика будет представлена в виде класса Grammar.

**Class Grammar{**

**beginning\_rules()** // метод возвращает итератор, указывающий на начало набора правил грамматики;

**start\_terms()** // метод возвращает итератор, указывающий на начало работы термов;

**ending\_rules()** // метод возвращает итератор, указывающий на конец набора правил грамматики;

**end\_terms()** // метод возвращает итератор, указывающий на конец набора термов грамматики;

**void import(string program)** // метод создаёт объект из строки

**get\_rules()** // метод возвращает набор правил вывода для грамматики;

**get\_starting\_symbol()** // метод возвращает начальный символ грамматики;

**get\_terms()** // метод возвращает список термов (терминалов, нетерминалов, операционных символов) грамматики;

**get\_terminals()** // метод возвращает список терминальных символов грамматики;

**get\_nonterminals()** // метод возвращает список нетерминальных символов грамматики;

**bool is\_terminal(Term t)** // метод возвращает true, если терм t принадлежит множеству терминалов грамматики;

**bool is\_nonterminal(Term t)** // метод возвращает true, если терм t принадлежит множеству нетерминалов грамматики;

**void set\_rules(Rules rules)** // метод задаёт набор правил вывода для грамматики;

**void set\_start\_symbol(Terms S)** // метод задаёт начальный символ грамматики;

**void set\_terms(Terms terms)** // метод задаёт список термов (терминалов, нетерминалов, операционных символов) грамматики;

}

Лексический анализатор представлен в виде класса Lexical\_Analyzer.

**Class Lexical\_Analyzer{**

**bool analyze(String prog)** // метод выполняет алгоритм лексического анализа входной строки prog с учетом ключевых слов и лексем

}

Алгоритм правого разбора входной цепочки реализован в классе Parser.

**Класс Parser{**

**Parser (Grammar g, Lexical\_Analyzer lex)** // конструктор класса, принимающий в параметрах входную грамматику g и лексический анализатор lex и инициализирует пустой стек pi;

**<unsigned int> pi** // стек номеров правил, получаемый в процессе правого разбора цепочки;

**State [] states** // список разбора входной строки, полученный выполнением алгоритма Эрли;

**parse(string program)** // метод выполняет лексический анализ входной строки program, алгоритм Эрли для построения списка разбора;

**R(State[] states, Situation s, unsigned int k)** // рекурсивная процедура правого разбора цепочки по списку разбора;

**make\_tree** // метод строит дерево вывода по правому разбору входной цепочки, находящемуся в стеке pi и выводит его, печатая в консоль;

}

Класс Parser взаимодействует с встроенным в Parser классом Early, который строит список разбора I0,…,In для входной цепочки длины n и заданной грамматики [1].

**Класс Earley{**

**start()** // метод запускает алгоритм разбора входной цепочки. возвращает список разбора;

**set\_starting\_state()** // начальное формирование. Метод задает множество начальных ситуаций в состоянии I0;

**bool is\_dot\_last(Situation s)** // метод, возвращает true, если метасимвол является последним символом в ситуации s;

**bool is\_dot\_before\_nonterminal(Situation s)** // метод, возвращает true, если метасимвол в ситуации s находится перед нетерминальным символом;

**add\_situation(State state, Situation s)** // метод, добавляет ситуацию s в состояние state;

**check\_comparison(State s)** // метод проводит сопоставление всех возможных терминалов в состоянии s и возвращает множество ситуаций для добавления в следующее состояние;

}

Для работы класса Early используются структуры State и Situation, передаваемые в параметры методов класса Early.

Структура Situation представляет собой объект вида [A ⟶ x1x2…xk ● xk+1…xm, i], называемый ситуацией, относящейся к входной цепочке, где A ⟶ x1…xm принадлежит множеству правил грамматики и 0≤i≤n, где n – длина входной цепочки.

**Struct Situation**{

**unsigned int k** // номер, принадлежащий данной ситуации;

**array<Term> before\_dot** // упорядоченный список термов, стоящих до метасимвола в ситуации;

**array<Term> after\_dot** // упорядоченный список термов, стоящих после метасимвола в ситуации;

**left\_part** // левая часть в ситуации;

**unsigned int dot\_position** // позиция метасимвола в ситуации;

}

Состояние рабочей среды алгоритма есть множество ситуаций. На каждом шаге алгоритма строится очередное состояние рабочей среды.

Последовательность I0…In состояний State называется списком разбора для входной цепочки.

**Struct State**{

**unsigned int i** // номер данного состояния

**set<Situation> situation\_set** // множество ситуаций данного состояния.

**add\_situation(Situation s)** // метод, добавляет ситуацию s в состояние;

}

Алгоритм правого разбора входной цепочки

***Шаг 1***. Построить список разбора входной цепочки по алгоритму Эрли при помощи метода start вложенного класса Early.

***Шаг 2.*** Если в состоянии In нет ситуации вида [S ⟶ α●, 0], где S – аксиома грамматики входного языка, то входная цепочка не принадлежит языку данной грамматики. Возвратить сообщение об ошибке и завершить работу алгоритма. В противном случае выполнить рекурсивную процедуру R([S ⟶ α●, 0], n).

***Шаг 3.*** В случае успешного завершения процедуры R, метод parse возвратит массив содержимого стека pi, в котором находится правый разбор входной цепочки. По данному массиву строится дерево разбора. которое выводится в консоль.

Построение списка разбора (алгоритм Эрли)

Правила построения состояния I0:

***Шаг 1.*** Начальное формирование. Включить в состояние I0 все ситуации вида [S ⟶ ●α, 0], где правило S ⟶ α принадлежит грамматике.

***Шаг 2.*** Успешное сопоставление нетерминала. Если [B →γ •,0], ∈I0, то для всех ситуаций [A→α • Bβ,0], уже принадлежащих I0, включить в I0 ситуацию [A→αB • β,0].

***Шаг 3.*** Разрастание дерева. Пусть ситуация [A→α • Bβ,0], ∈I0. Для каждого правила грамматики вида B →γ включить в I0 ситуацию [B →•γ ,0], если такой ситуации в I0 ещё нет.

Пункты 2 и 3 следует повторять, пока новые ситуации могут быть добавлены в состояние I­0.

Правила построения состояния Ij:

***Шаг 4***. Выполнение всех возможных сопоставлений терминалов. Для каждой ситуации [B →α • aβ,i], для которой a = aj (j-ый символ входной цепочки), включить в Ij ситуацию [B →αa • β,i].

***Шаг 5.*** Успешное сопоставление нетерминала. Допустим, что [A→α• ,i] ∈Ii. Искать в Ii ситуации вида [B →γ • Aβ,k] . Для каждой из них включить в Ij ситуацию [B →γA• β,k].

***Шаг 6.*** Разрастание дерева. Допустим, что [A→α • Bβ,k] ∈Ij. Для каждого правила грамматики B →γ включить в Ij ситуацию [B →•γ , j].

Пункты 5 и 6 применяются, пока новые ситуации могут быть добавлены в состояние Ij. Алгоритм Эрли заключается в построении Ij, где 0 n, n – длина входной цепочки.

Определение процедуры R([A ⟶ β●, i], j)

***Шаг 1.*** Если A ⟶ β – правило грамматики с номером h, добавить в стек pi номер правила h.

***Шаг 2.*** Если β=X1X2…Xm, то положить k = m и c = j. Если k=0, то остановиться.

***Шаг 3. А)***  Если терм Xk принадлежит множеству терминалов грамматики, то положить k=k-1, c=c-1.

***Шаг 3. Б)*** Если терм Xk принадлежит множеству нетерминалов грамматики, то найти в Ic ситуацию [Xk ⟶ γ●, r], где некоторый r таков, что [A ⟶ x1x2…xk ● xk+1…xm, i] ∈ Ir. Затем выполнить R([Xk ⟶ γ●, r], c), положить k=k-1, c=r.

***Шаг 4.*** Повторять шаг 3, пока k ≠ 0. При k = 0 остановиться.

# Проект алгоритма правого разбора

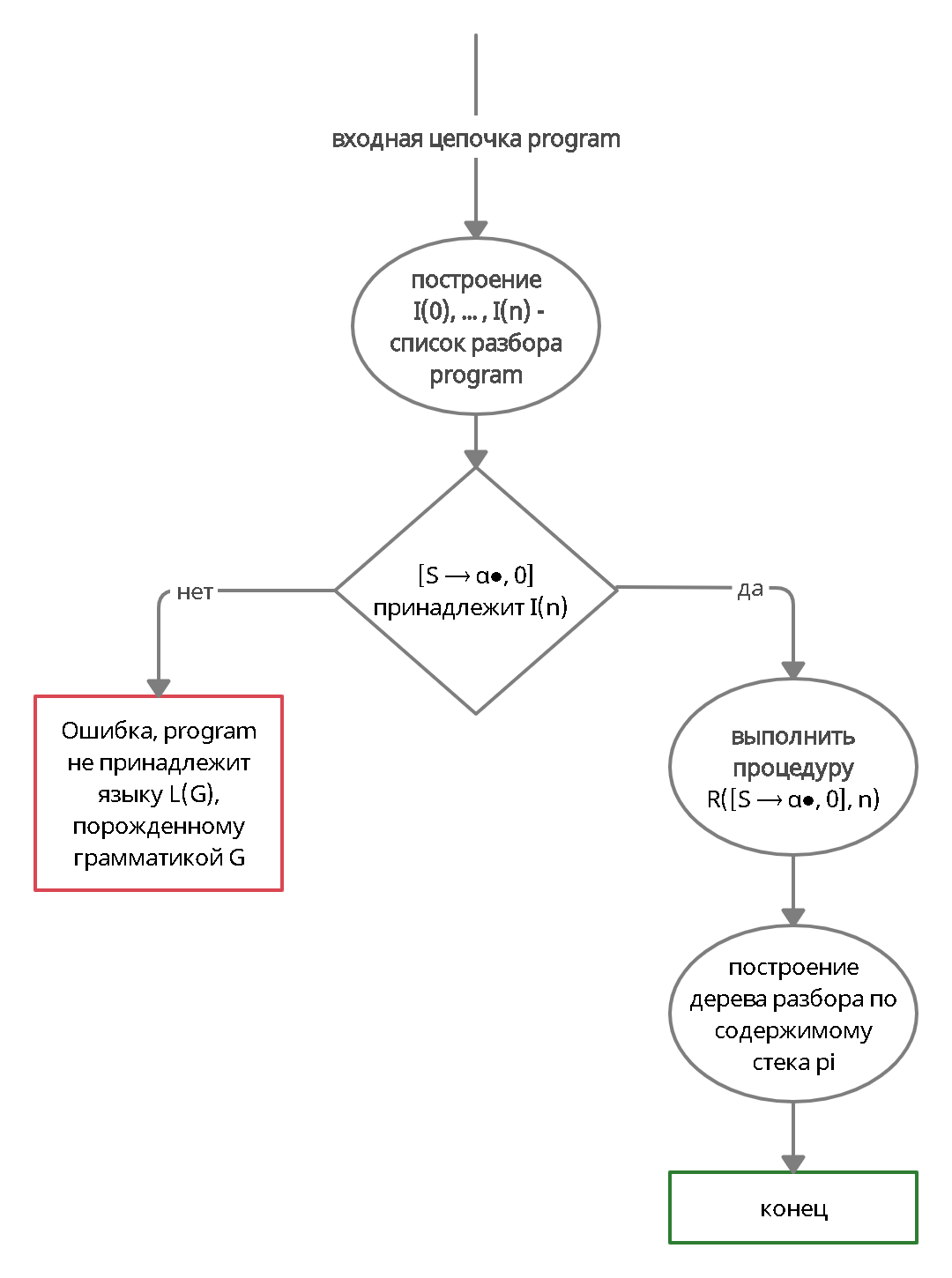


Рисунок 2 – Блок-схема правого разбора

# Синтаксический анализатор

Синтаксический анализатор получает на вход последовательность лексем (входная цепочка). При помощи алгоритма Эрли строится список разбора, затем выполняется алгоритм правого разбора входной цепочки. Результатом правого разбора является стек pi, содержащий последовательность номеров применяемых правил грамматики.

Если ошибок не возникло, то по содержимому стека pi строится дерево разбора.

# Пример дерева разбора для следующей входной цепочки

Пусть входная цепочка имеет вид:

Package com.company;

import java.lang.Math;

public class Main{

public static void main (String[] args) {

int a = 15;

System.out.println(a);

}

}

Пример дерева разбора данной цепочки представлен на рисунке 3:

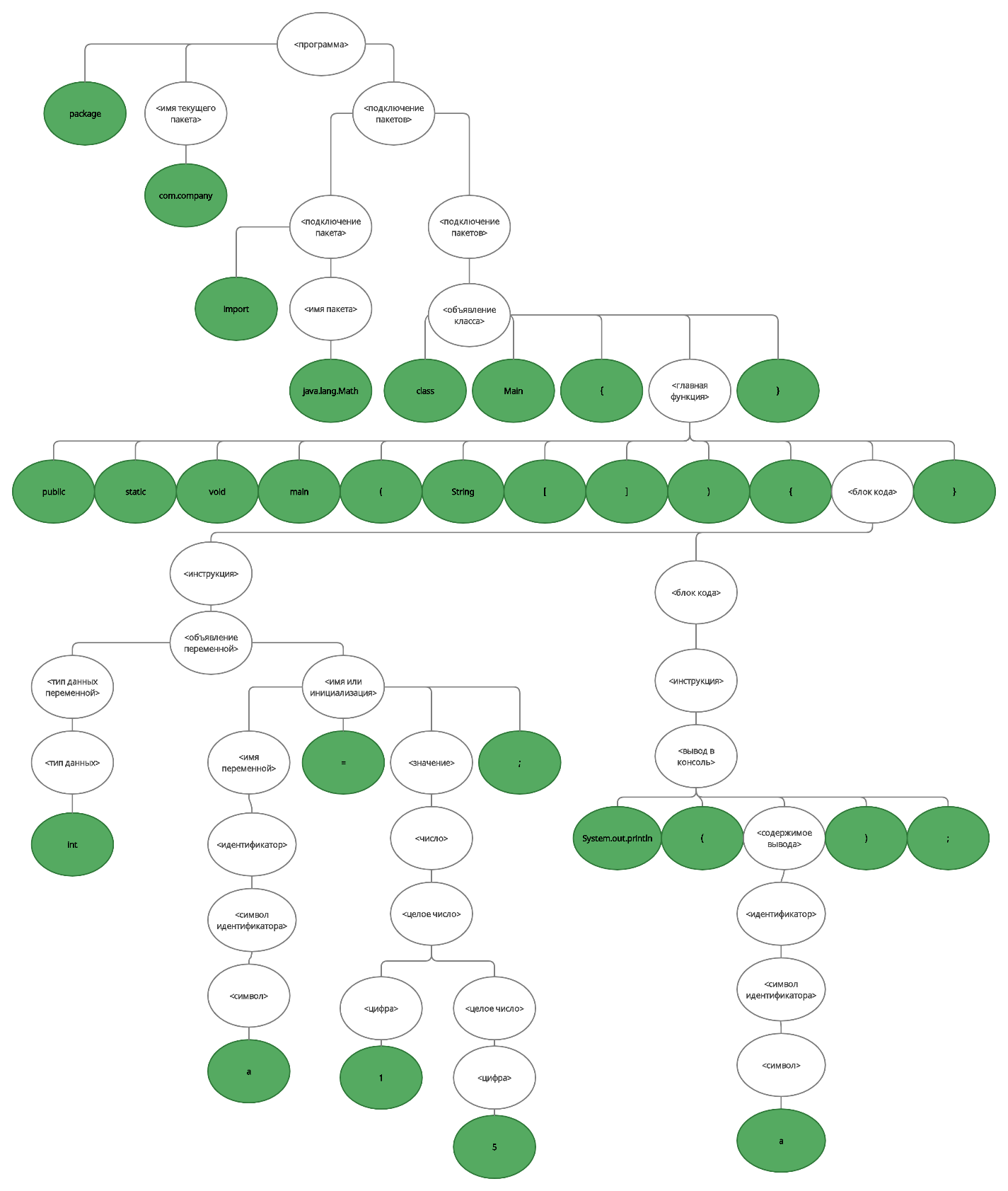


Рисунок 3 - Дерево разбора

# Ошибки синтаксического анализатора

Процесс синтаксического анализа может обнаружить ошибку, если встречена лексема, но не имеется продукции с этой лексемой в правой части в правилах грамматики, т.е. если встреченная лексема не может быть выведена в грамматике входного языка.

Таким образом, в процессе синтаксического анализа могут быть встречены ошибки:

• Ошибка 0: ожидалось имя пакета

• Ошибка 1: ожидался главный класс с названием Main

• Ошибка 2: ожидалось тело класса

• Ошибка 3: ожидалась главная функция main

• Ошибка 4: ожидался тип данных переменной

• Ошибка 5: ожидалось имя переменной

• Ошибка 6: ожидалось значение

• Ошибка 7: ожидался тип данных функции

• Ошибка 8: ожидалось имя функции

• Ошибка 9: ожидалось тело функции

• Ошибка 10: ожидался идентификатор

• Ошибка 11: ожидалось выражение

• Ошибка 12: ожидалось логическое выражение

• Ошибка 13: ожидался логический операнд

• Ошибка 14: ожидался побитовый операнд

# Заключение

В ходе курсового проекта были решены следующие задачи:

1. Сформулирована неформальная постановка задачи.

2. Разработана грамматика подмножества входного языка Java.

3. Описаны контекстные условия языка Java.

4. Создан проект лексического анализатора, включающий в себя таблицы соответствий, ключевых и зарезервированных слов, типов лексем, конечный автомат.

5. Создан проект синтаксического анализатора, включающий в себя алгоритм синтаксического анализа, представление грамматики, проект алгоритма, пример дерева разбора для входной цепочки, анализ ошибок синтаксического анализатора.

# Список литературы

1. Ахо, А. В. Компиляторы: принципы, технологии и инструментарий компиляторов / А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман. – М. : Вильямс, 2008. – 1184 с.

URL: <https://e.lanbook.com/book/1298>

2. Артемьева, И. Л. Теория вычислительных процессов и структур. Часть 1. Языки и способы их задания. Учебно-методическое пособие / И. Л. Артемьева. – Владивосток : Издательство Дальневосточного федерального университета, 2011. – 60 с.

3. Самоучитель Java [электронный ресурс].

URL: <http://proglang.su/java>

**Исполнители работы**

В работе над курсовым проектом принимали участие студенты группы Б8118-09.03.04прогин: Полеся В.А., **Скопецкий А.Г**. Участниками коллектива выполнялись следующие задания:

|  |  |
| --- | --- |
| **Фамилия И.О.** | **Разделы/подразделы** |
| Полеся В.А. | 2. Синтаксис входного языка  3. Контекстные условия  4. Таблица соответствия  6. Проект синтаксического анализатора  Заключение  Список литературы |
| Скопецкий А.Г. | Введение  1. Неформальная постановка задачи  2. Синтаксис входного языка  3. Контекстные условия  5. Конечный автомат лексического анализатора |