|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Поволжский Государственный Университет Телекоммуникаций и Информатики** Кафедра «ИВТ»   |  |  | | --- | --- | | Сдана на проверку  «\_\_\_» \_\_\_\_\_\_\_\_ 2024 г. | Допустить к защите  «\_\_\_» \_\_\_\_\_\_\_\_ 2024 г.  Защищена с оценкой  «\_\_\_» \_\_\_\_\_\_\_\_ 2024 г. |   **КУРСОВАЯ РАБОТА** По дисциплине: «Теория автоматов и формальных языков» На тему: «Разработка компилятора модельного языка»  Пояснительная записка   |  |  | | --- | --- | | Студент группы ИВТ-93 \_\_\_\_\_\_\_\_\_\_ Барсков Н. А.  (подпись) (ФИО)  Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Бахарева Н.Ф.  (подпись) (ФИО) | 190070 (№ зачетной книжки) |   Самара, 2024 |

**Рецензия**

Содержание

[**Введение** 4](#_Toc122917993)

[**1.** **Постановка задачи** 5](#_Toc122917994)

[**2.** **Формальная модель задачи** 7](#_Toc122917995)

[**2.1. Формальные грамматики** 7](#_Toc122917996)

[**2.2. Формы Бэкуса-Наура** 8](#_Toc122917997)

[**2.3. Расширенные формы Бэкуса-Наура** 8](#_Toc122917998)

[**2.4. Диаграммы Вирта** 10](#_Toc122917999)

[**3. Спецификация основных процедур и функций** 19](#_Toc122918000)

[**3.1. Лексический анализатор** 19](#_Toc122918001)

[**3.2. Диаграмма состояний с действиями** 25](#_Toc122918002)

[**3.3. Синтаксический анализатор** 28](#_Toc122918003)

[**3.4. Семантический анализ** 29](#_Toc122918004)

[**4. Работа с программным средством** 31](#_Toc122918005)

[**4.1. Укрупненная схема алгоритма программного средства** 31](#_Toc122918006)

[**4.2. Детальная разработка алгоритмов отдельных подзадач** 32](#_Toc122918007)

[**4.2.1. Блок-схема лексического анализатора** 32](#_Toc122918008)

[**4.2.2. Блок-схема синтаксического анализатора** 50](#_Toc122918012)

[**4.2.3. Блок-схема семантического анализатора** 51](#_Toc122918013)

[**5. Работа с программным средством** 52](#_Toc122918014)

[**5.1. Описание интерфейса** 52](#_Toc122918015)

[**5.2. Перечень ошибок** 53](#_Toc122918016)

[**5.3. Пример работы компилятора** 54](#_Toc122918017)

[**Заключение** 64](#_Toc122918018)

[**Список использованных источников** 65](#_Toc122918019)

[**Приложение А – Текст программы** 66](#_Toc122918020)

# **Введение**

Компилятор – это программа, которая создаётся для перевода программы, написанной на высокоуровневом языке программирования в машинный код, который в последствие будет исполняться процессором компьютера.

В большинстве случаев компиляция программы происходит полностью. Компилятор целиком считывает программу, проводит её пошаговый анализ (лексический, синтаксический, семантический), оптимизирует её, очищая от излишних конструкций, но сохраняя исходный смысл операций, и также целиком переводит её в машинный код.

Цель курсовой работы:

- закрепление теоретических знаний в области теории формальных языков, грамматик, автоматов и методов трансляции;

- формирование практических умений и навыков разработки собственного компилятора модельного языка программирования;

- закрепление практических навыков самостоятельного решения инженерных задач, развитие творческих способностей студентов и умений пользоваться технической, нормативной и справочной литературой.

В настоящее время в мире появляются более новые языки программирования и не каждый из ныне существующих трансляторов могут прочитать программы, написанный на новом языке, и перевести его в другой язык. Поэтому сейчас разрабатываются новые трансляторы, в этом и заключается актуальность данной курсовой работы.

# **Постановка задачи**

Разработать компилятор модельного языка, выполнив следующие действия.

1) В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

а) РБНФ;

б) диаграмм Вирта;

в) формальных грамматик.

2) Написать пять содержательных примеров программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих примерах все его функциональные возможности.

3) Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.

4) По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.

5) Разработать программное средство, реализующее лексический анализ текста программы на входном языке.

6) Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.

7) Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.

8) Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.

9) Распечатать пример таблиц идентификаторов и двуместных операций.

10) Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.

11) Составить набор контрольных примеров, демонстрирующих:

а) все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке;

б) представить ход интерпретации синтаксически и семантически правильной программы с помощью таблицы.

# **Формальная модель задачи**

Существуют три основных метода описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура и диаграммы Вирта.

**2.1. Формальные грамматики**

**Определение.** Формальной грамматикой называется четверка вида:

, (1.1)

где *VN* - конечное множество нетерминальных символов грамматики (обычно прописные латинские буквы);

*VT* - множество терминальных символов грамматики (обычно строчные латинские буквы, цифры, и т.п.), *VT* *VN =*;

*Р* – множество правил вывода грамматики, являющееся конечным подмножеством множества (*VT VN*)*+ * (*VT VN*)*\**; элемент (*, *) множества *Р -* называется правилом вывода и записывается в виде **** (читается: «из цепочки ** выводится цепочка **»);

*S* – начальный символ грамматики, *S* *VN*.

Для записи правил вывода с одинаковыми левыми частями вида используется сокращенная форма записи .

**Пример:**

P→ program D B.

*D* → *var | var D1*

D1 → I | I D1

I → I1:T;

I1 → I2|I1,I2

I2 → C|I2C|I2R

B → begin O end

O → O1|O;O1

O1 → S|A|F|W|Y|U|V|ничего

S → O1|SS1O1

S1 → :|\n

A → I2 as E

F → F1|F1 else O1

F1 → if E then O1

W → for A to E do O1

Y → while E do O1

U → read(I1)

V → write(E1)

E1 → E|E1,E

E → Z|EZ1Z

Z → J|ZJ1J

J → M|JM1M

M → I2|N|L|~M|(E)

L → true|false

M1 → mult|div|and

J1 → plus|min|or

Z1 → NE|EQ|LT|LE|GT|GE

N1 → N|Nd|N*D*

N2 → N21N23

N21 →N22|N22N21

N22 → 0|1

N23 → B|b

N8 → N81N82

N81 → N83|N83N81

N82 → O|o

N83 →0|1|2|3|4|5|6|7

N16 →N161N162

N161 →R|N161RN163

N162 →H|h

N163 → A|B|C|D|E|F|a|b|c|d|e|f

Nf →Nf1|Nf2

Nf1 →NNp

Nf2 →Nf21|Nf21Np|.N|.Nf1

Nf21→N.N

Np→ Ne|NeNs

Ne→E|e

Ns→+|-

N → R|RN

T → int|float|bool

C → A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

*R* → *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*

## **2.2. Формы Бэкуса-Наура**

Метаязык, предложенный Бэкусом и Науром, использует следующие обозначения:

- символ «::=» отделяет левую часть правила от правой (читается: «определяется как»);

- нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;

- терминалы — это символы, используемые в описываемом языке;

- правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|» (читается: «или»).

**2.3. Расширенные формы Бэкуса-Наура**

Для повышения удобства и компактности описаний, в РБНФ вводятся следующие дополнительные конструкции (метасимволы):

- квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;

- фигурные скобки «{» и «}» означают повторение заключенной в них синтаксической конструкции ноль или более раз;

- сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз;

- круглые скобки «(» и «)» используются для ограничения альтернативных конструкций.

***Вариант: 311311***

<операции\_группы\_отношения>:: = *NE | EQ | LT | LE | GT | GE*

<операции\_группы\_сложения>:: = *plus | min | or*

<операции\_группы\_умножения>:: =  *mult | div | and*

<унарная\_операция>::= ~

<выражение>::= <операнд>{<операции\_группы\_отношения> <операнд>}

<операнд>::= <слагаемое> {<операции\_группы\_сложения> <слагаемое>}

<слагаемое>::= <множитель> {<операции\_группы\_умножения> <множитель>}

<множитель>::= <идентификатор> | <число> | <логическая\_константа> |

<унарная\_операция> <множитель> | (<выражение>)

<число>::= <целое> | <действительное>

<логическая\_константа>::= true | false

<идентификатор>::= <буква> {<буква> | <цифра>}

<буква>::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |

U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |

p | q | r | s | t | u | v | w | x | y | z

<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<целое>::= <двоичное> | <восьмеричное> | <десятичное> |

<шестнадцатеричное>

<двоичное>::= {/ 0 | 1 /} (B | b)

<восьмеричное>::= {/ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 /} (O | o)

<десятичное>::= {/ <цифра> /} [D | d]

<шестнадцатеричное>::= <цифра> {<цифра> | A | B | C | D | E | F | a | b |

c | d | e | f} (H | h)

<действительное>::= <числовая\_строка> <порядок> |

[<числовая\_строка>] . <числовая\_строка> [порядок]

<числовая\_строка>::= {/ <цифра> /}

<порядок>::= ( E | e )[+ | -] <числовая\_строка>

<программа>::= program var <описание> begin <оператор> {; <опера-

тор>} end.

<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}

<тип>::= int | float | bool

<оператор>::= <составной> | <присваивания> | <условный> |

<фиксированного\_цикла> | <условного\_цикла> | <ввода> |

<вывода>

<составной>::= <оператор> { ( : | перевод строки) <оператор> }

<присваивания>::= <идентификатор> as <выражение>

<условный>::= if <выражение> then <оператор> [ else <оператор>]

<фиксированного\_цикла>::= for <присваивания> to <выражение> do

<оператор>

<условного\_цикла>::= *while <выражение> do <оператор>*

<ввода>::= *read «(» <идентификатор> {, <идентификатор> }«)»*

<вывода>::= *write «(»<выражение> {, <выражение> }«)»*

|  |  |
| --- | --- |
| Признак начала комментария | Признак конца комментария |
| { | } |

## **2.4. Диаграммы Вирта**

В метаязыке диаграмм Вирта используются графические примитивы.

При построении диаграмм учитывают следующие правила:

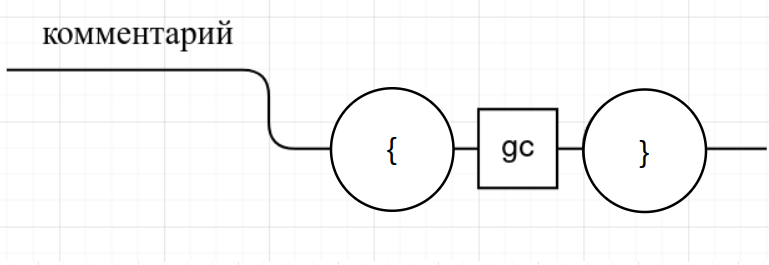
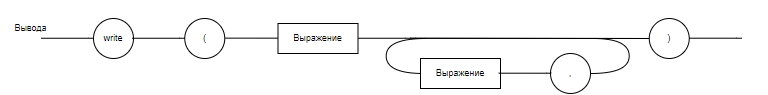
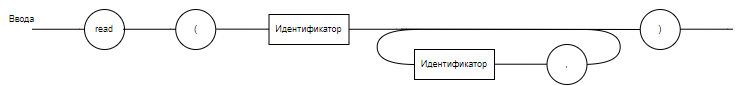
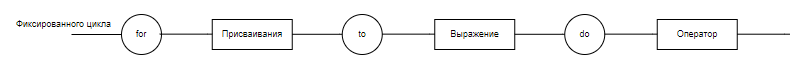
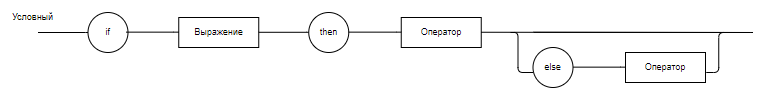
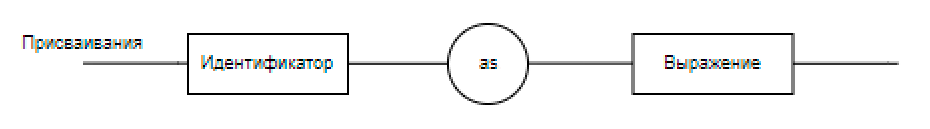
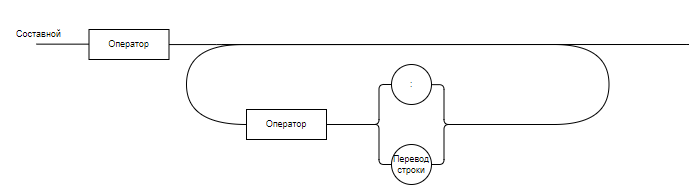
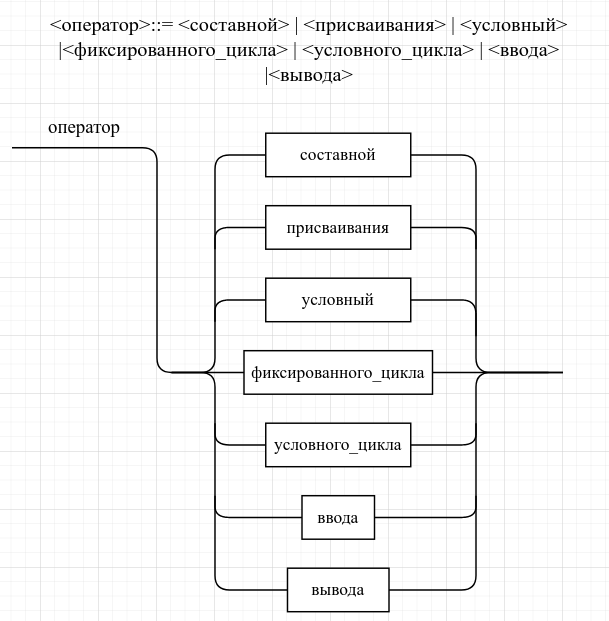
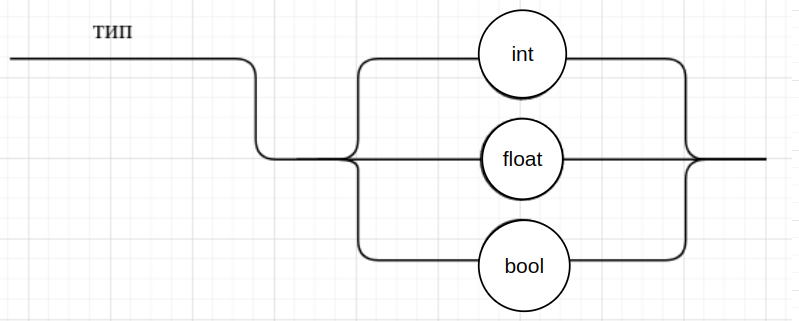
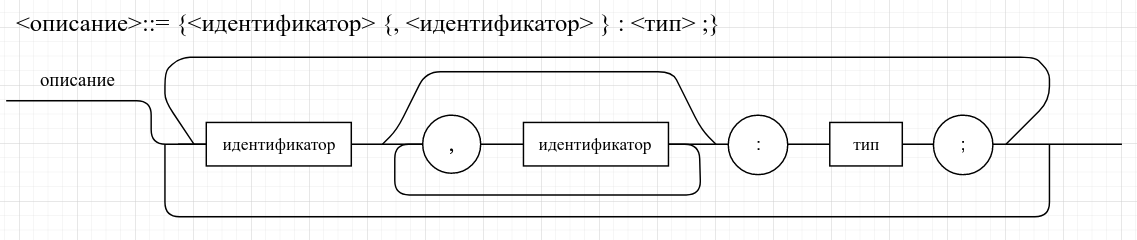
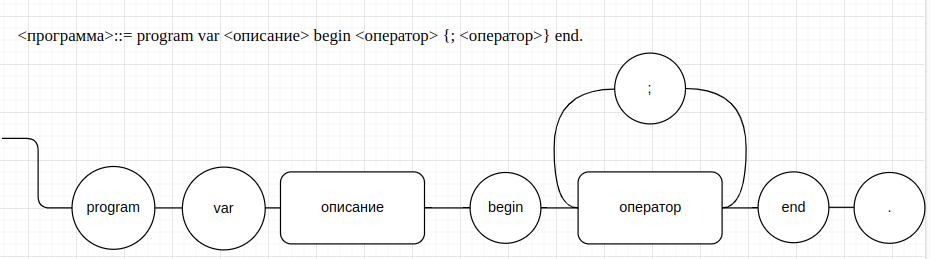
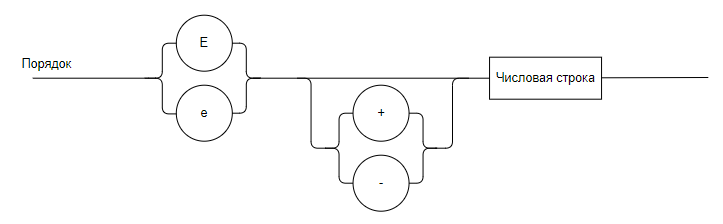
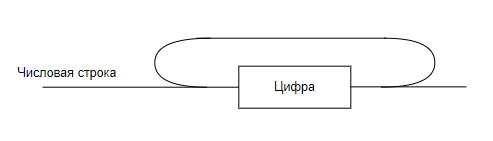
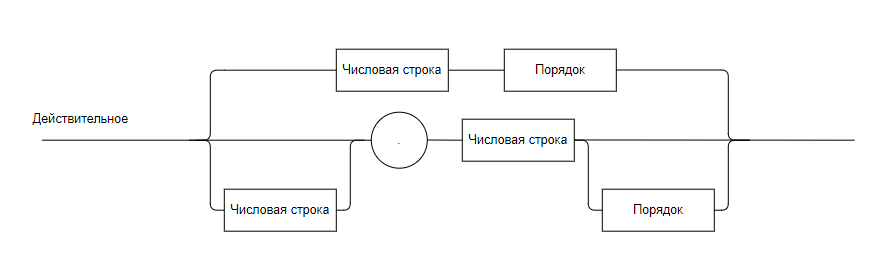
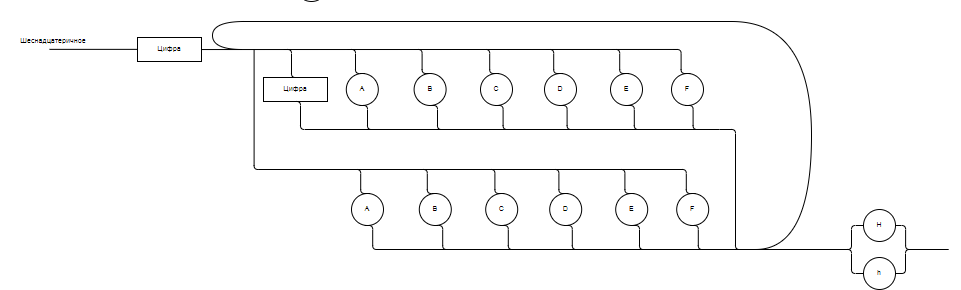
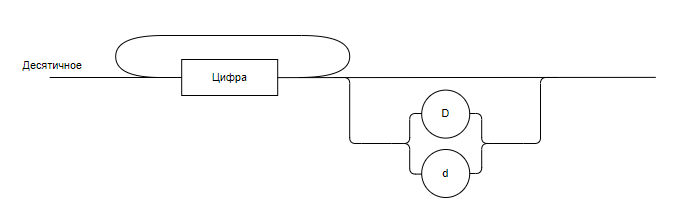
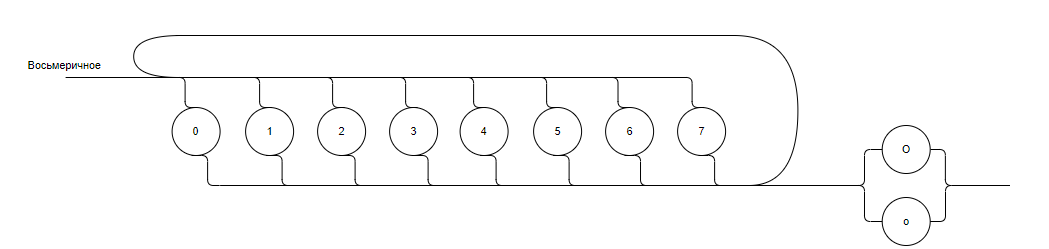
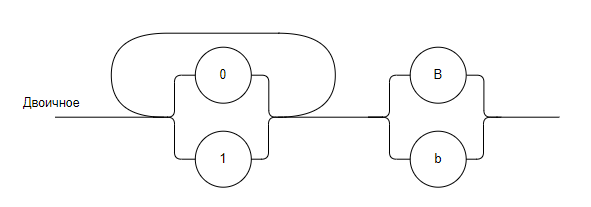
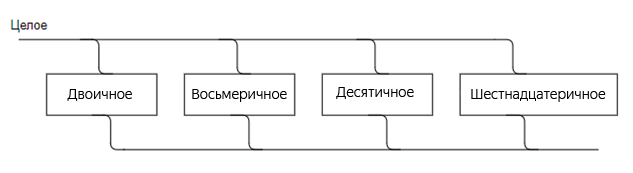
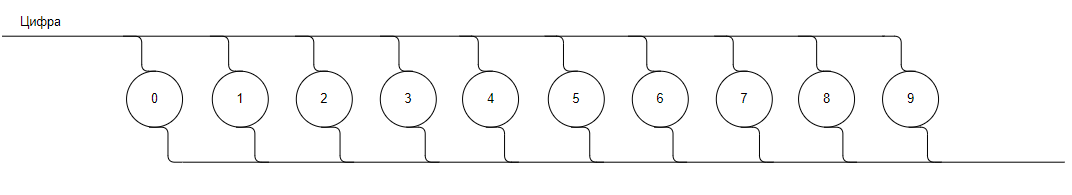
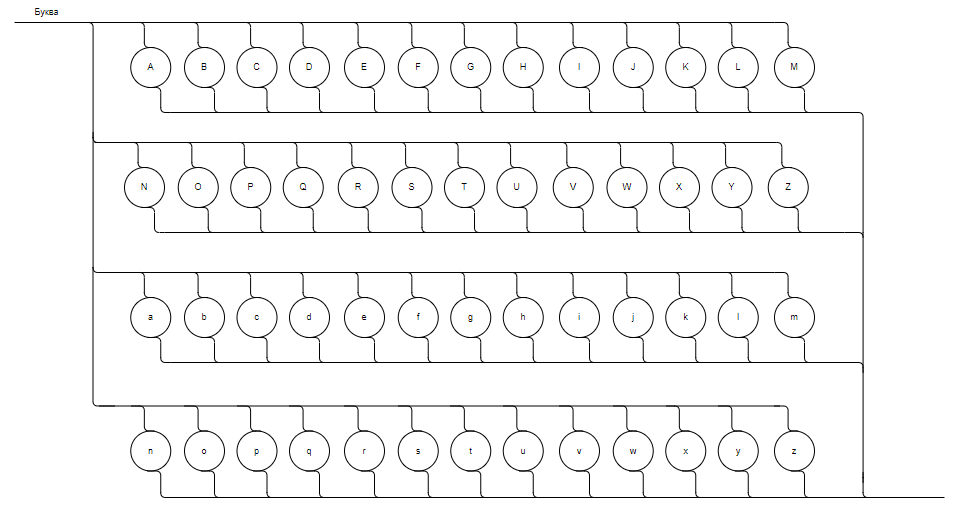
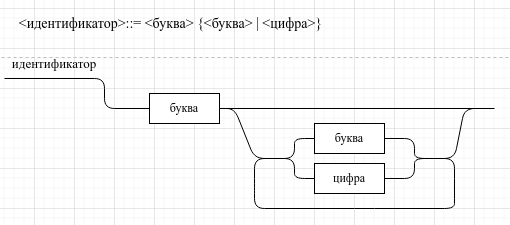
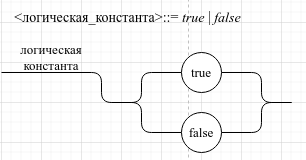
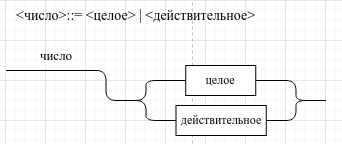
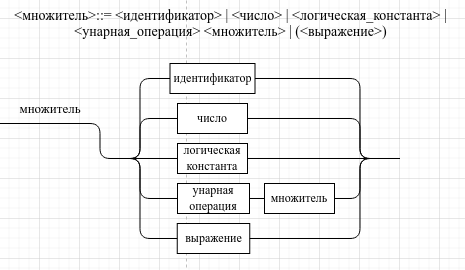
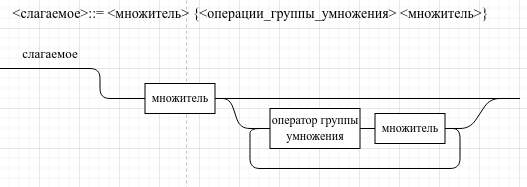
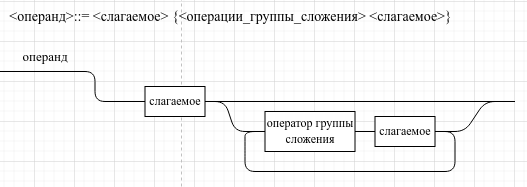
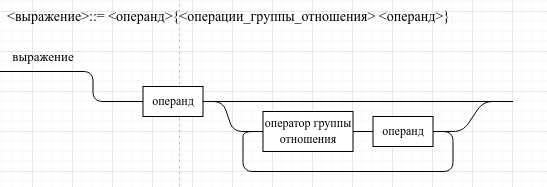
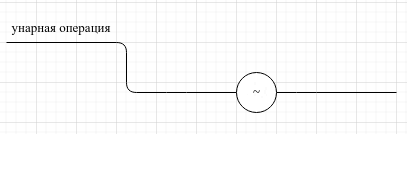
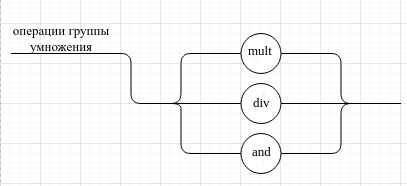
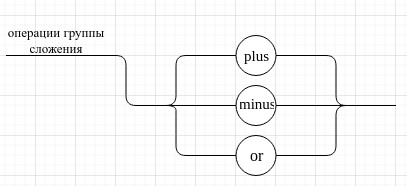
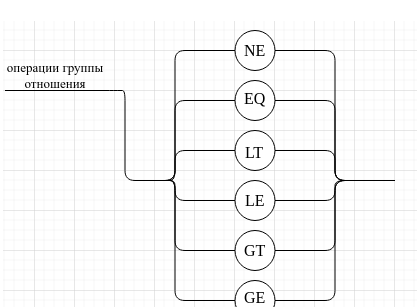
- каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно изображаются на противоположных сторонах;

- каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;

- альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;

- должна быть одна входная дуга (располагается обычно слева или сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу);

- стрелки на дугах диаграмм обычно не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений.



# **3. Спецификация основных процедур и функций**

## **3.1. Лексический анализатор**

Лексический анализатор (ЛА) – это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются в отдельные минимальные единицы текста, несущие смысловую нагрузку – лексемы.

Задача лексического анализа - выделить лексемы и преобразовать их к виду, удобному для последующей обработки.

Каждая лексема представляет собой пару чисел вида (n, k), где n – номер таблицы лексем, k - номер лексемы в таблице.

Входные данные ЛА - текст транслируемой программы на входном языке.

Выходные данные ЛА - таблица лексем в числовом представлении.

Структура данных, выбранная для хранения таблиц – является List <>, т.е. таблицы: ключевых слов, разделителей, идентификаторов, чисел – хранятся в виде списков.

Списки – представляют собой удивительно гибкий инструмент по работе с коллекциями. Одной из главных особенностей списков является возможность использовать любой тип данных. В списках удобно хранить и управлять данными, т. к. списки позволяют манипулировать указателями. Следовательно, т.к. есть возможность манипулировать указателями – скорость работы с данными списков очень высока, а значит разрабатываемый компилятор модельного языка М, будет работать очень быстро

Поиск в таблице реализован методом двоичного поиска. Бинарный (двоичный) поиск – это алгоритм поиска индекса элемента в упорядоченном массиве (списке). На каждой итерации происходит деление массива на две части. Поскольку на каждой итерации количество элементов в рабочей области массива уменьшается вдвое, метод бинарного поиска является очень эффективным.

Запись лексем происходит следующим образом: после того, как встроенный метод для списков Contains () вернул True (т.е. найдено совпадение с одной из таблиц лексем), вызываем метод Add () для списка лексем. Данный метод добавляет в конец списка лексему, состоящую из 2 цифр: первая цифра – номер таблицы лексемы, которая вернула True, 2 цифра – порядковый номер значения таблицы, которое вернула True.

Таблица № 0 – таблица ключевых слов:

|  |  |  |  |
| --- | --- | --- | --- |
| Ключевые слова | | | |
| 1 | *int* | 10 | *to* |
| 2 | *float* | 11 | *do* |
| 3 | *bool* | 12 | *while* |
| 4 | *if* | 13 | *read* |
| 5 | *then* | 14 | write |
| 6 | *else* | 15 | *true* |
| 7 | *for* | 16 | *false* |
| 8 | *program* | 17 | *end* |
| 9 | *var* | 18 | *begin* |

Таблица № 1 – таблица разделителей:

|  |  |  |  |
| --- | --- | --- | --- |
| Разделители | | | |
| 1 | NE | 12 | mult |
| 2 | EQ | 13 | div |
| 3 | LT | 14 | and |
| 4 | LE | 15 | ~ |
| 5 | GT | 16 | as |
| 6 | GE | 17 | ; |
| 7 | plus | 18 | : |
| 8 | min | 19 | , |
| 9 | or | 20 | ) |
| 10 | { | 21 | ( |
| 11 | } | 22 | . |

Таблицы идентификаторов (3) и чисел (4) формируются в ходе лексического анализа.

Алгоритм ЛА:

Считываем символ введенной программы:

* Если это буква – считываем буквы и цифры, пока не встретим разделитель или ключевое слово. Если цепочка символов введенной программы, совпадает с одним из ключевым словом, то записываем результат в таблицу лексем (1, z). Если цепочка символов введенной программы, совпадает с одним из разделителей, то записываем результат в таблицу лексем (2, z). Если это не ключевое слово и не разделитель, то является идентификатором. Считываем после идентификатора его разделитель. Записываем идентификатор и его разделитель, через двоеточие, в таблицу идентификаторов, и записываем лексему идентификатора с разделителем (3, z);
* Если это цифра – считываем пока не встретим разделитель. Проверяем числа на правильность. Если число введено верно, то переводим числа в двоичное представление. Записываем число и его бинарное представление в таблицу чисел, и записываем лексему числа (4, z)
* Если это символ – считываем пока не найдем совпадение с таблицей разделителей. Если это разделитель, то записываем лексему разделителя в таблицу лексем (2, z)
* Если это неизвестный символ, то выдаем ошибку.

Опишем результаты работы лексического анализатора для модельного языка *М*.

Входные данные ЛА:

program var a,b : int; c, d: float; e, f: bool;

begin a as 5

b as 7: e as 7.5: d as c: e as true: f as false

b as b min a

if b LT a then a as a plus b else b as b min 5

end.

Выходные данные ЛА:

program: KEY\_WORD

var: KEY\_WORD

a: IDENTIFIER

,: DELIMETER

b: IDENTIFIER

:: DELIMITER

int: KEY\_WORD

;: DELIMITER

c: IDENTIFIER

,: DELIMETER

d: KEY\_WORD

:: DELIMETER

float: KEY\_WORD

;: DELIMITER

e: IDENTIFIER

,: DELIMETER

f: KEY\_WORD

bool: KEY\_WORD

;:DELIMETER

begin: KEY\_WORD

a: KEY\_WORD

as: DELIMETER

5: NUMBER

b: IDENTIFIER

as: DELIMETER

7: NUMBER

::DELIMETER

e: IDENTIFIER

as: DELIMETER

7.5: NUMBER

::DELIMETER

d: IDENTIFIER

as: DELIMETER

c: IDENTIFIER

::DELIMETER

e: IDENTIFIER

as: DELIMETER

true: KEY\_WORD

::DELIMETER

f: IDENTIFIER

as: DELIMETER

false: KEY\_WORD

b: IDENTIFIER

as: DELIMETER

b: IDENTIFIER

min: DELIMETER

a: IDENTIFIER

if: KEY\_WORD

b: IDENTIFIER

LT:DELIMETER

a: IDENTIFIER

then: KEY\_WORD

a: IDENTIFIER

as: DELIMETER

a: IDENTIFIER

plus: DELIMETER

b: IDENTIFIER

else: KEY\_WORD

b: IDENTIFIER

as: DELIMETER

b: IDENTIFIER

min: DELIMETER

5: NUMBER

end: KEY\_WORD

.:DELIMETER

Составим ЛА для модельного языка. Предварительно введем следующие обозначения для переменных, процедур и функций.

Переменные:

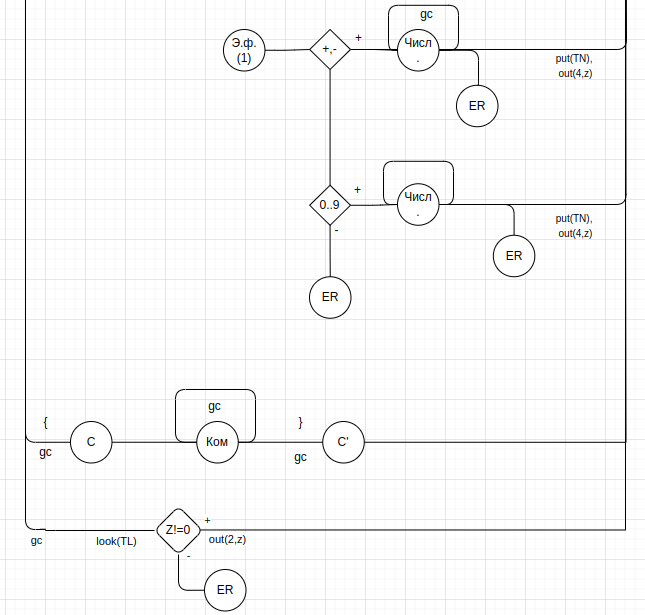
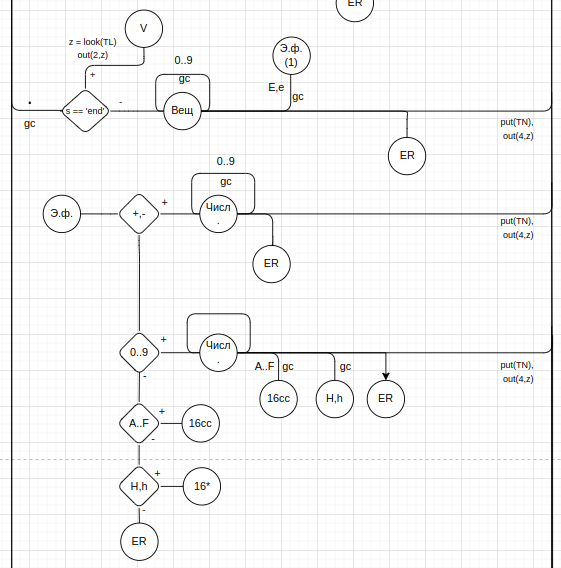
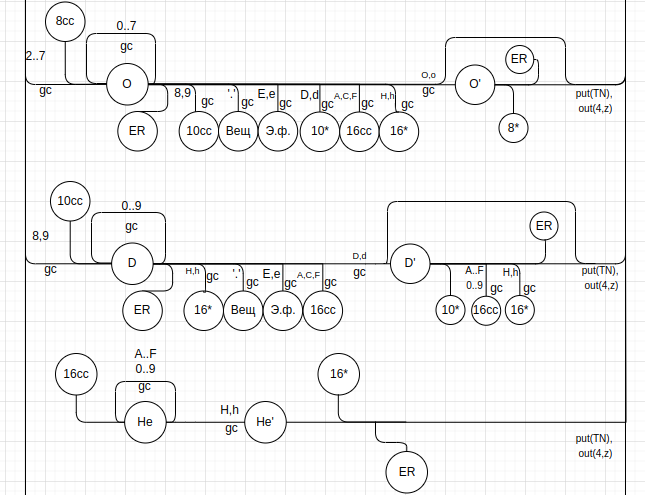
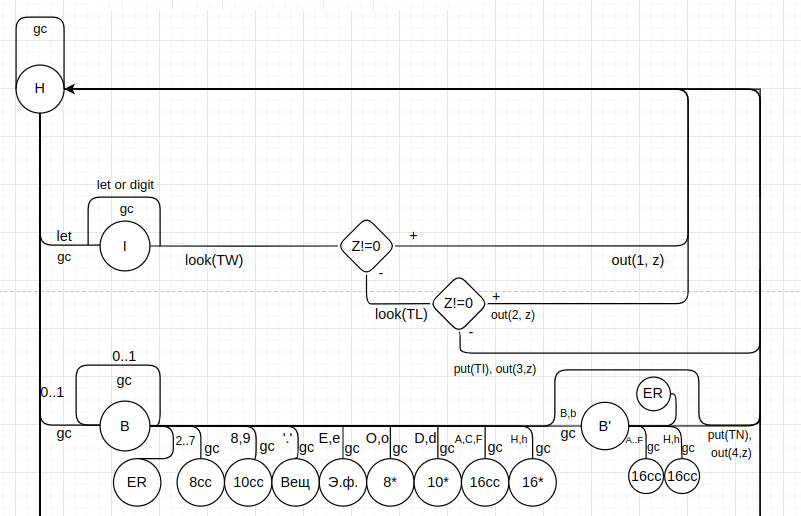
1. СН – очередной входной символ;
2. S – буфер для накапливания символов лексемы;
3. CS – текущее состояние буфера накопления лексем с возможными значениями: H – начало, I – идентификатор, N – число, С – комментарий, *SK –* считывание скобки, R – ограничитель, V – выход программы, *ER* – ошибка.
4. t — таблица лексем анализируемой программы с возможными значениями: TW – таблица ключевых слов М-языка, TL – таблица разделителей М-языка, TI – таблица идентификаторов программы, TN – таблица целых чисел;
5. z - номер лексемы в таблице t (если лексемы в таблице нет, то z=0).

Процедуры и функции:

1. gc – процедура считывания очередного символа текста в переменную СН;
2. let – логическая функция, проверяющая, является ли переменная СН буквой;
3. digit - логическая функция, проверяющая, является ли переменная СН цифрой;
4. flush – процедура очистки буфера S;
5. add – процедура добавления очередного символа в конец буфера S;
6. look(t) – процедура поиска лексемы из буфера *S* в таблице *t* с возвращением номера лексемы в таблице;
7. put(t) – процедура записи лексемы из буфера S в таблицу t, если там не было этой лексемы, возвращает номер данной лексемы в таблице;
8. out(n, z) – процедура записи пары чисел (n, z) в файл лексем.

Построим ДС с действиями для распознавания и формирования внутреннего представления лексем модельного языка *М*.

**3.2. Диаграмма состояний с действиями**



**3.3. Синтаксический анализатор**

Задача синтаксического анализатора: провести разбор текста программы, сопоставив его с эталоном, данным в описании языка. Для синтаксического разбора используются контекстно-свободные грамматики (КС-грамматика).

Один из эффективных методов синтаксического анализа – метод рекурсивного спуска.

В основе метода рекурсивного спуска лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям).

Метод рекурсивного спуска реализует разбор цепочки сверху вниз. Для каждого нетерминального символа грамматики создается своя процедура, носящая его имя. Задача этой процедуры начиная с указанного места исходной цепочки, найти подцепочку, которая выводится из этого нетерминала. Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибок, которая выдает сообщение о том, что цепочка не принадлежит языку грамматики и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры составляется непосредственно по правилам вывода, соответствующего нетерминала, при этом терминалы распознаются самой процедурой, а нетерминалам соответствуют вызовы процедур, носящих их имена.

В курсовой работе применяется такая проверка открытия и закрытия программы, как:

1. Код в программе начинается с разделителя «{»;

2. Считав разделитель «}» в конце, объявляется конец программы;

В качестве примера рассмотрен оператор условного цикла:

1. Считывание ключевого слова while;

3. Считывается выражение, если не выражение – выдаётся ошибка;

4. Конец выражения определяется ключевым словом do;

5. За ключевым словом do ожидается оператор, при его отсутствии выдаётся ошибка.

**3.4. Семантический анализ**

В ходе семантического анализа проверяются отдельные правила записи исходных программ, которые не описываются КС-грамматикой. Эти правила носят контекстно-зависимый характер, их называют семантическими соглашениями или контекстными условиями.

Соблюдение контекстных условий для языка М предполагает три типа проверок:

1) обработка описаний;

2) анализ выражений;

3) проверка правильности операторов.

Вход: файл лексем в числовом представлении.

Выход: заключение о семантической правильности программы или о типе обнаруженной семантической ошибки.

Задача обработки описаний – проверить, все ли переменные программы описаны правильно и только один раз.

Задача анализа выражений – проверить описаны ли переменные, встречающиеся в выражениях, и соответствуют ли типы операндов друг другу и типу операции.

Задачи проверки правильности операторов:

1) выяснить, все ли переменные, встречающиеся в операторах, описаны;

2) установить соответствие типов в операторе присваивания;

Задача решается проверкой типов в соответствующих местах программы.

Если при анализе цепочек лексем возникает ошибка или встретилась неизвестная лексема или лексема, не принадлежащая анализируемому оператору, то вызывается исключение с текстом ошибки.

**4.** **Работа с программным средством**

**4.1. Укрупненная схема алгоритма программного средства**



Рис. 1. Блок-схема алгоритма программного средства

**4.2. Детальная разработка алгоритмов отдельных подзадач**

### **4.2.1. Блок-схема лексического анализатора**

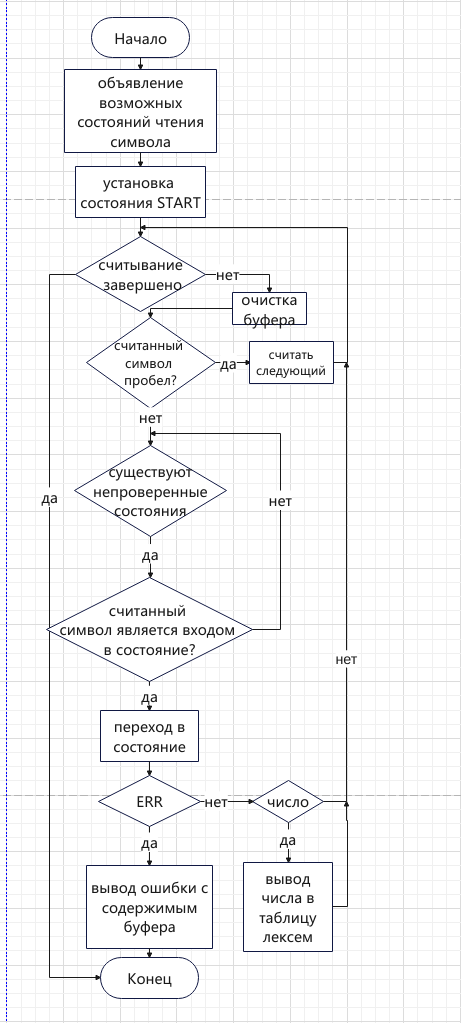


Рис.2.1 Обобщенная блок-схема лексического анализатора

### 



Рис.2.2 Блок-схема лексического анализатора в состоянии считывания идентификатора

### 

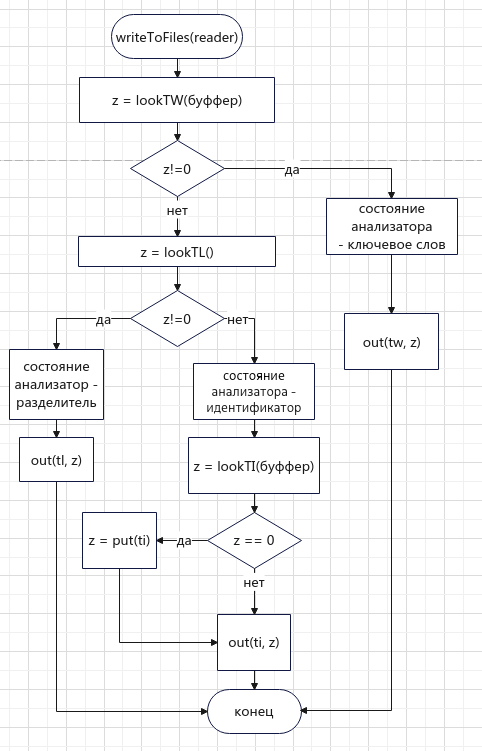


Рис.2.3 Блок-схема процедуры записи лексемы в файлы для состояния идентификатора

### 

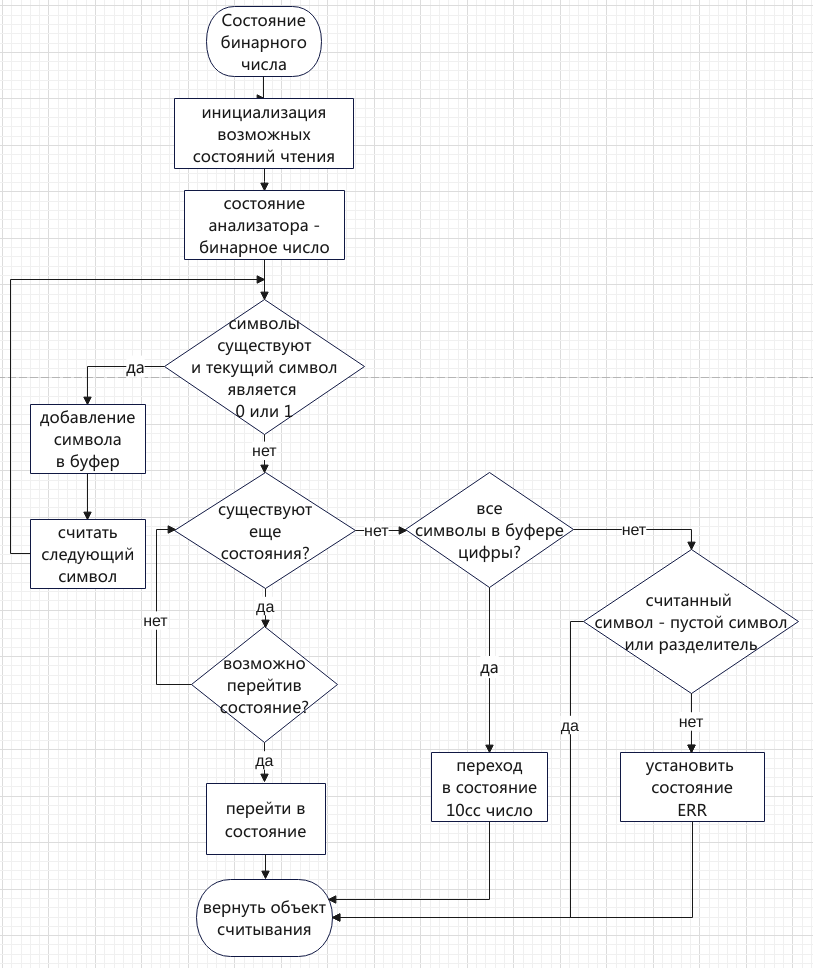


Рис.2.5 Блок-схема лексического анализатора в состоянии считывания бинарного числа

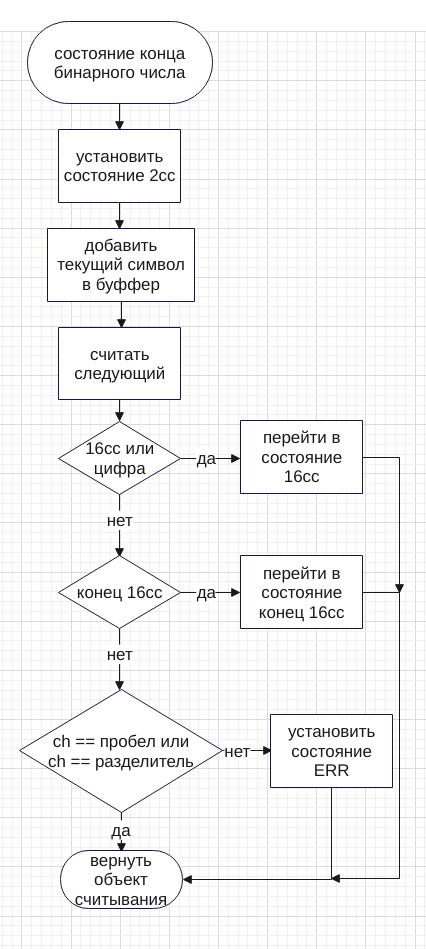


Рис.2.6 Блок-схема лексического анализатора в состоянии считывания конца бинарного числа

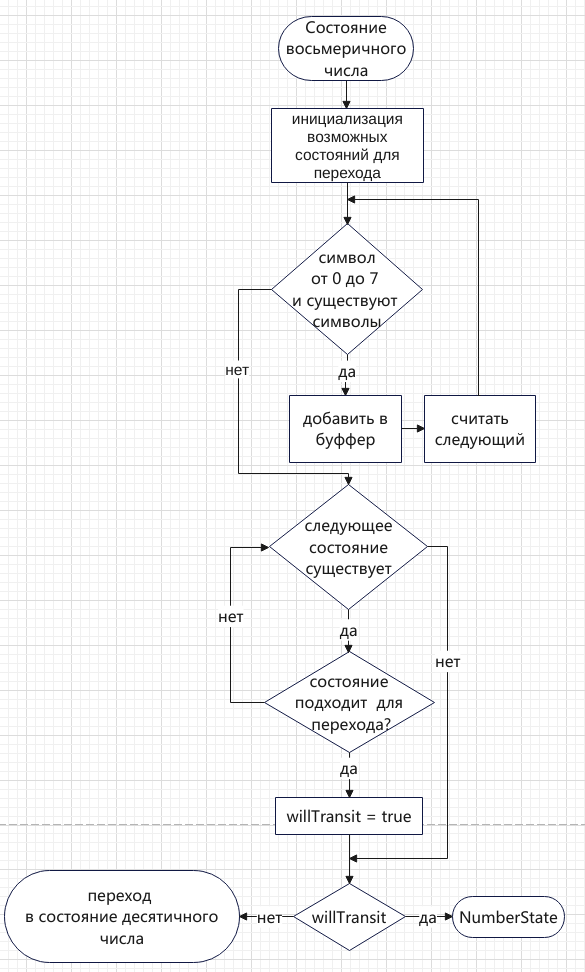


Рис.2.7. Блок-схема лексического анализатора в состоянии 8сс числа

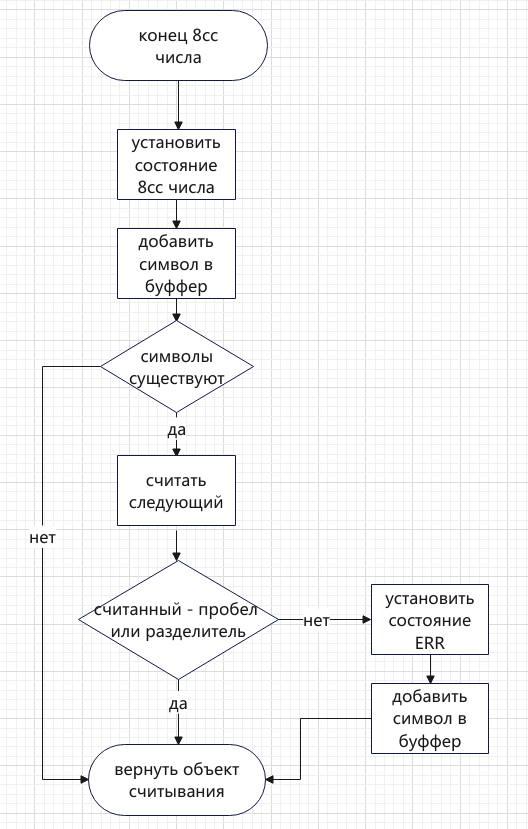


Рис.2.8. Блок-схема лексического анализатора в состоянии конца 8сс числа

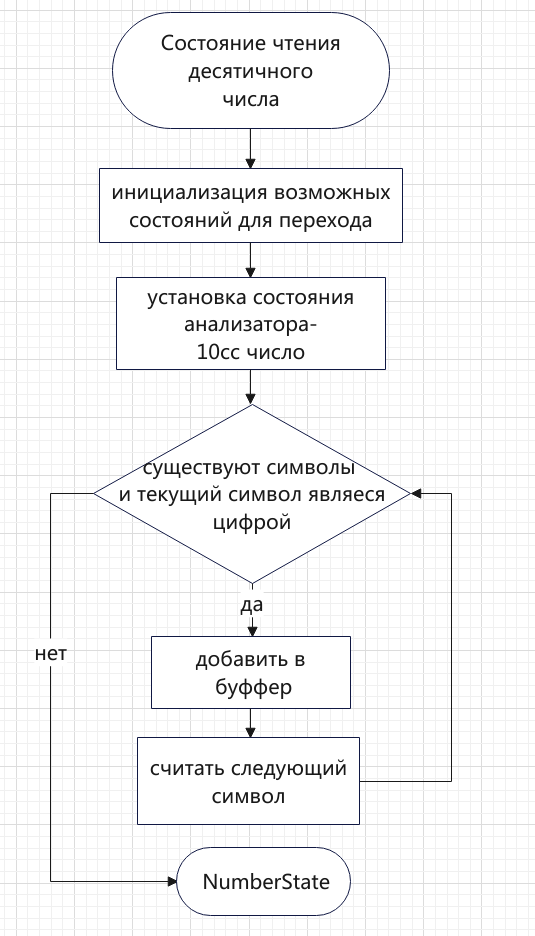


Рис.2.9. Блок-схема лексического анализатора в состоянии 10сс числа

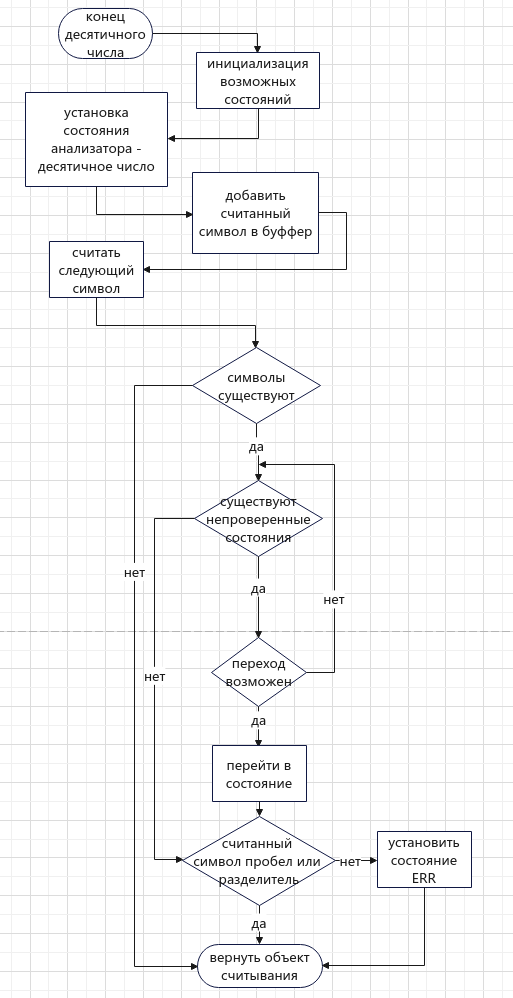


Рис.2.10. Блок-схема лексического анализатора в состоянии конца 10сс числа

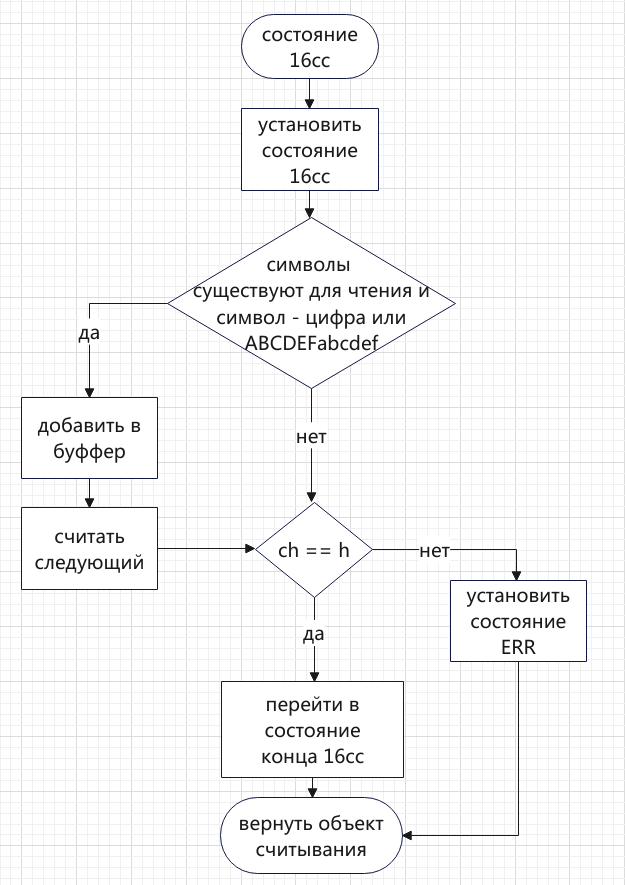


Рис.2.11. Блок-схема лексического анализатора в состоянии 16сс числа

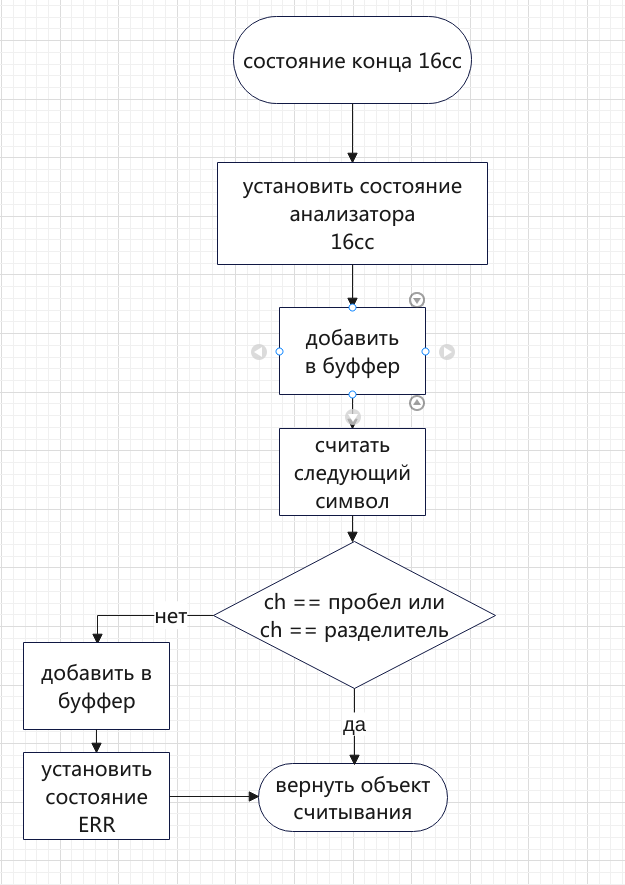


Рис.2.12. Блок-схема лексического анализатора в состоянии конца 16сс числа

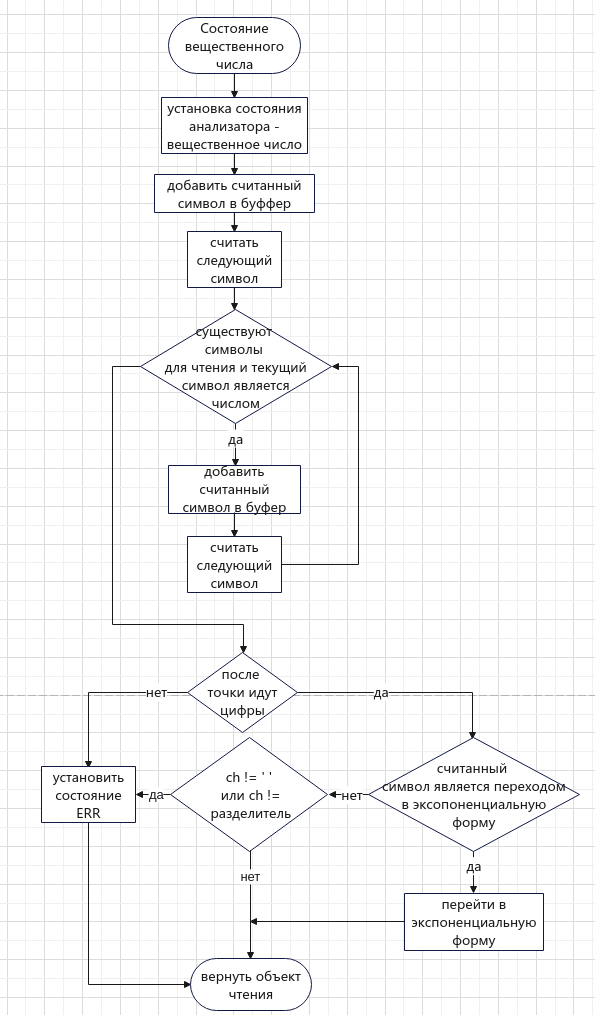


Рис.2.13. Блок-схема лексического анализатора в состоянии вещественного числа

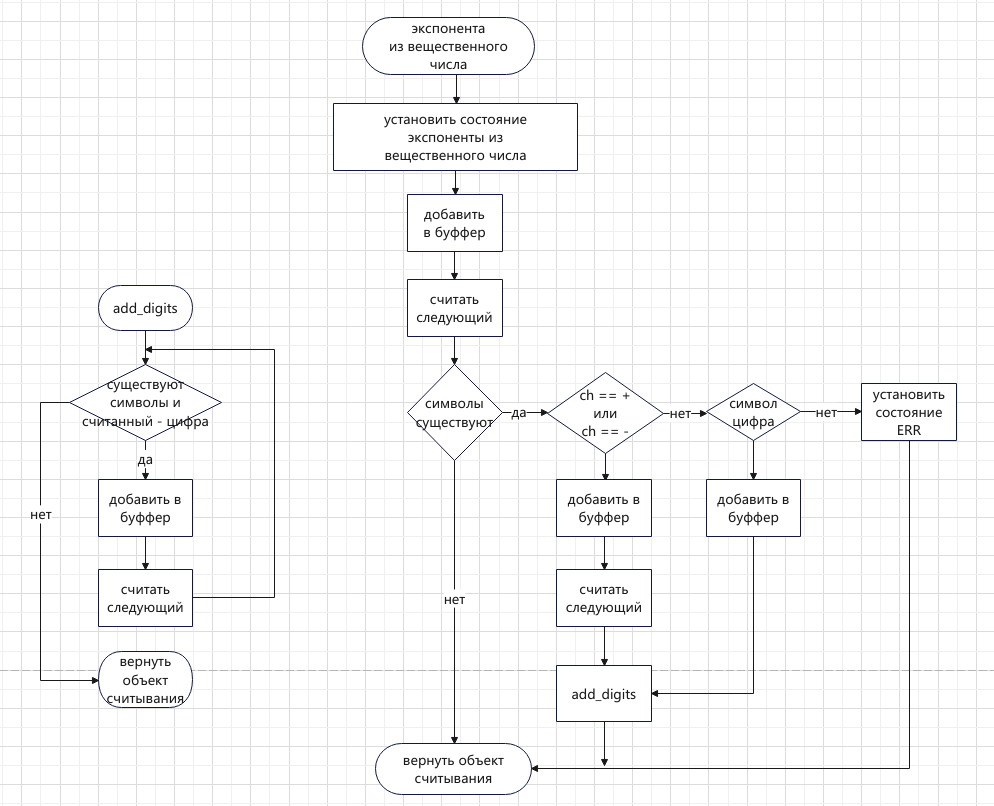


Рис.2.14. Блок-схема лексического анализатора в состоянии экспоненты из вещественного числа

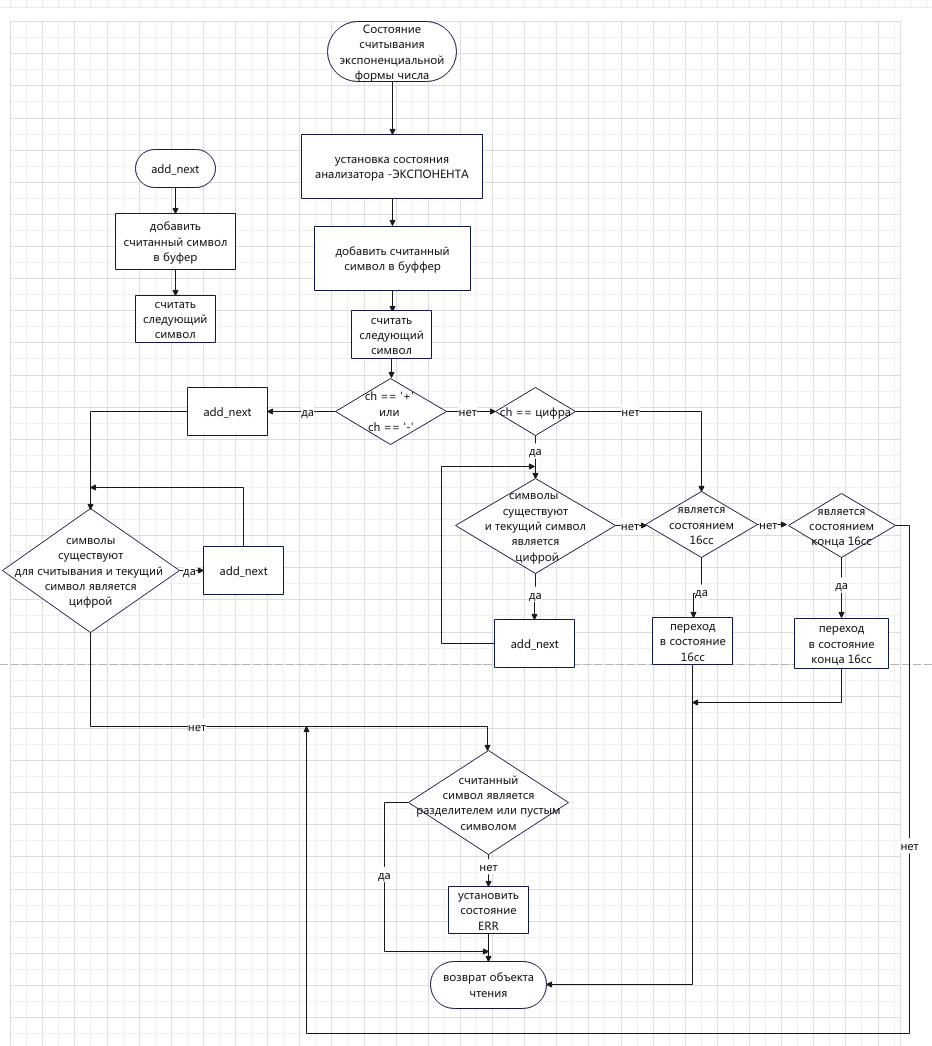


Рис.2.14. Блок-схема лексического анализатора в состоянии считывания экспоненты

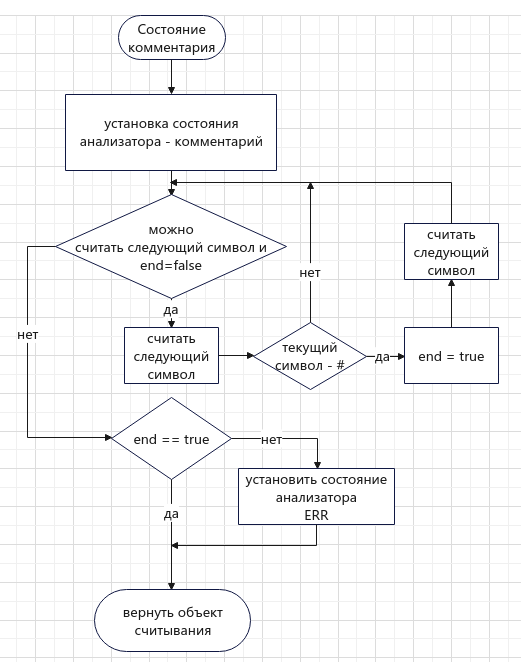


Рис.2.15. Блок-схема лексического анализатора в состоянии считывания комментария

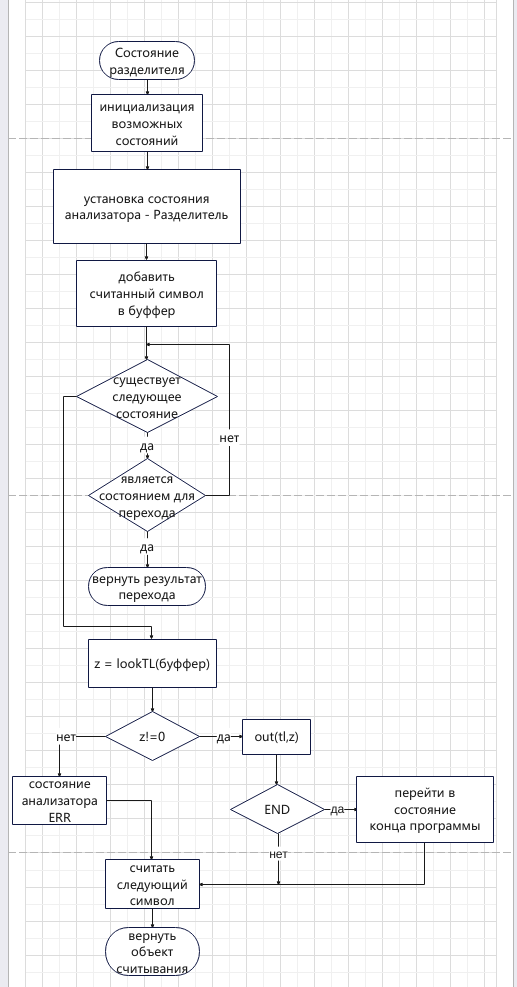


Рис.2.15. Блок-схема лексического анализатора в состоянии считывания разделителя

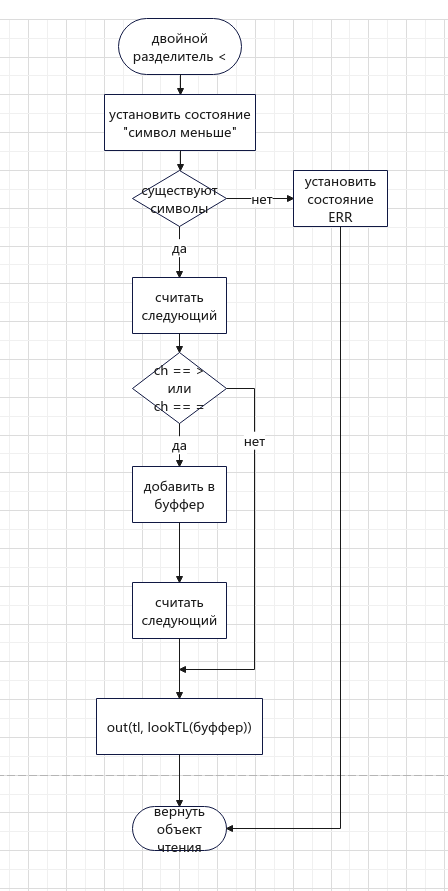


Рис.2.16. Блок-схема лексического анализатора в состоянии считывания разделителя начинающего на „<“

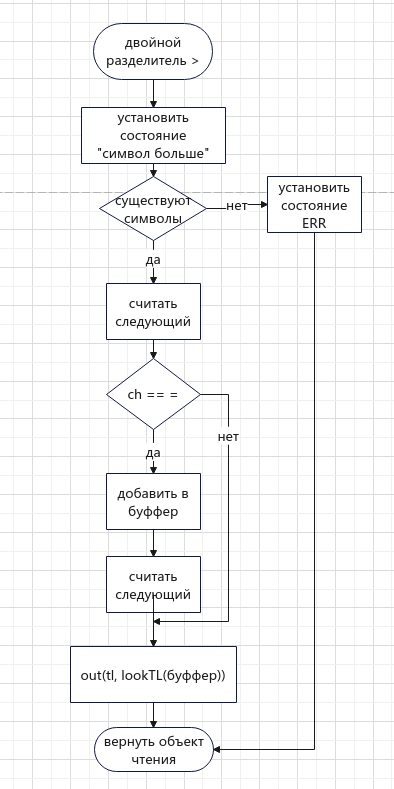


Рис.2.17. Блок-схема лексического анализатора в состоянии считывания разделителя начинающего на „>“

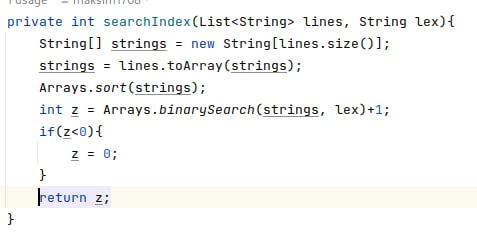


Рис.2.18. Реализация бинарного поиска лексем

### **4.2.2. Блок-схема синтаксического анализатора**

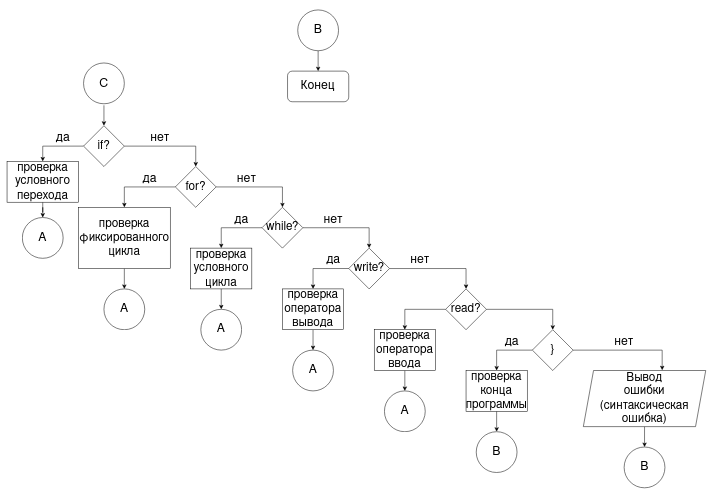
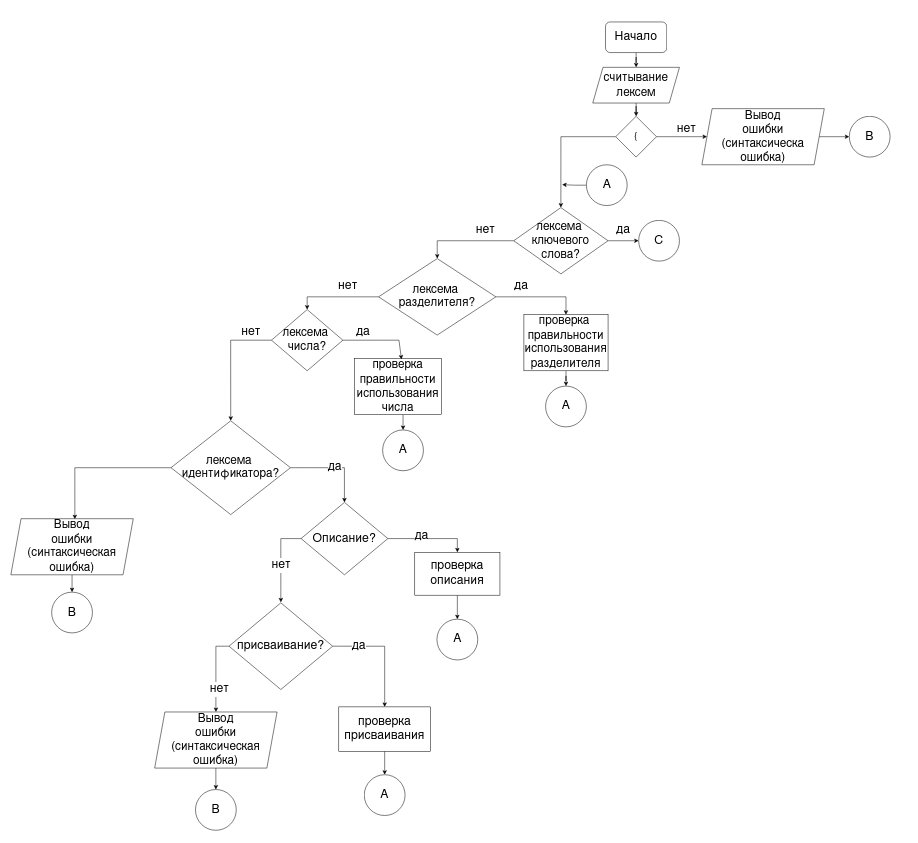


Рис.3. Блок-схема синтаксического анализатора

### **4.2.3. Блок-схема семантического анализатора**

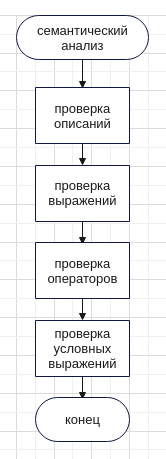


Рис.4. Блок-схема семантического анализатора

**5. Работа с программным средством**

## **5.1. Описание интерфейса**

Примеры работы программы на разных наборах исходного кода проиллюстрированы на скриншотах в части 5.2.

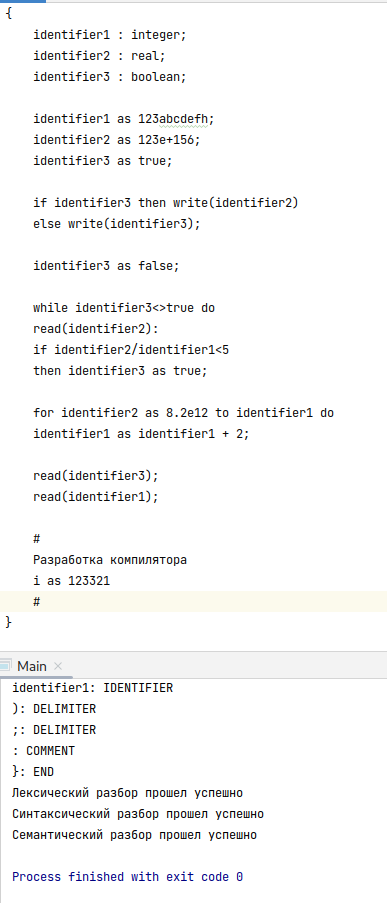


Рис.5. Главный интерфейс

## **5.2. Перечень ошибок**

Перечень ошибок:

1. Лексические ошибки:
   1. Неправильный формат числа
2. Синтаксические ошибки:
   1. Нарушен порядок, ожидалось ключевое слово
   2. Нарушен порядок, ожидался разделитель
   3. Нарушен порядок, ожидался идентификатор
3. Семантические ошибки:
   1. Переменная уже существует
   2. Нельзя применять операнды типа boolean по отношению к переменным типа integer
   3. Нельзя применять операнды типа boolean по отношению к переменным типа real
   4. Нельзя применять операнды численного типа по отношению к переменным типа boolean
   5. Оператор while требует наличие только boolean выражения в условии
   6. Оператор for требует наличие только boolean выражения в условии
   7. Оператор условия требует наличие только boolean выражения
   8. Вы не можете присвоить значение типа <тип1> к переменной типа <тип2>

## **5.3. Пример работы компилятора**

Правильный пример:

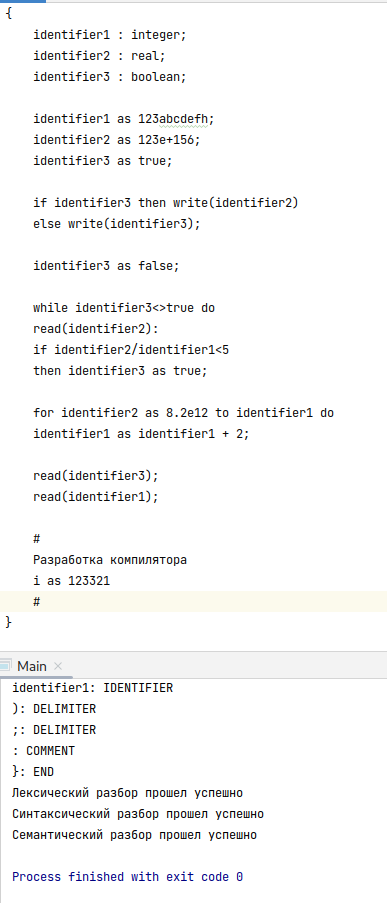


Рис.6. – Правильный пример работы компилятора

1) Ошибка: «Неправильный формат числа»

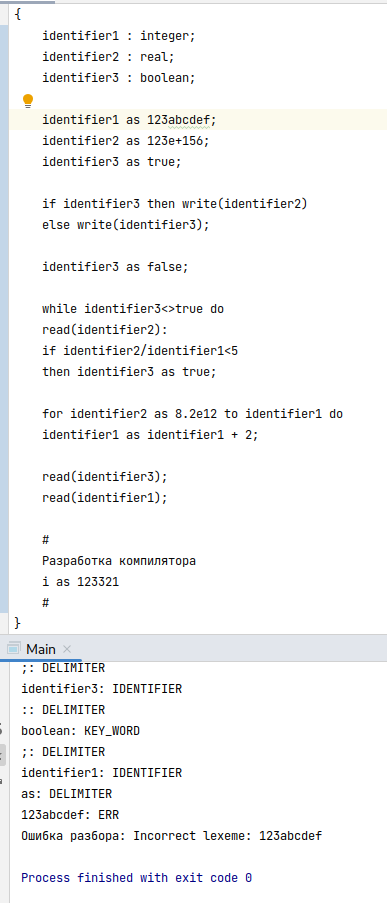


Рис.7. Ошибка: «Неправильный формат числа»

2) Ошибка: «Нарушен порядок, ожидалось ключевое слово/разделитель»

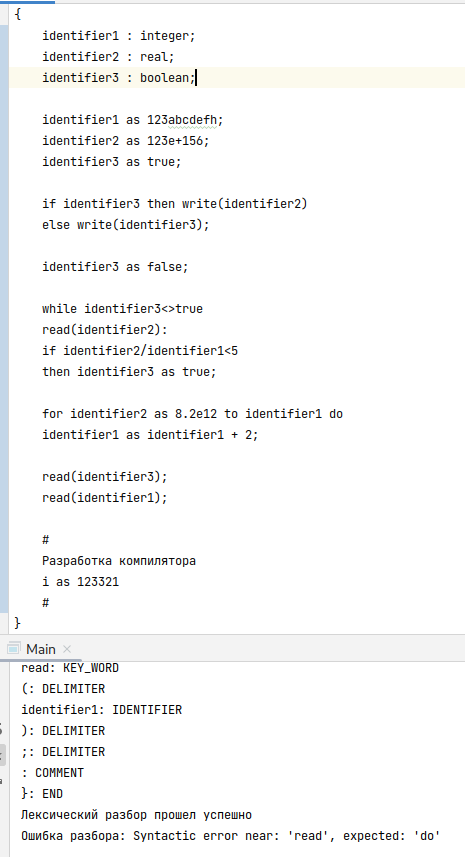


Рис.8. Ошибка: «Нарушен порядок, ожидалось ключевое слово/разделитель»

3) Ошибка: «Переменная уже существует»

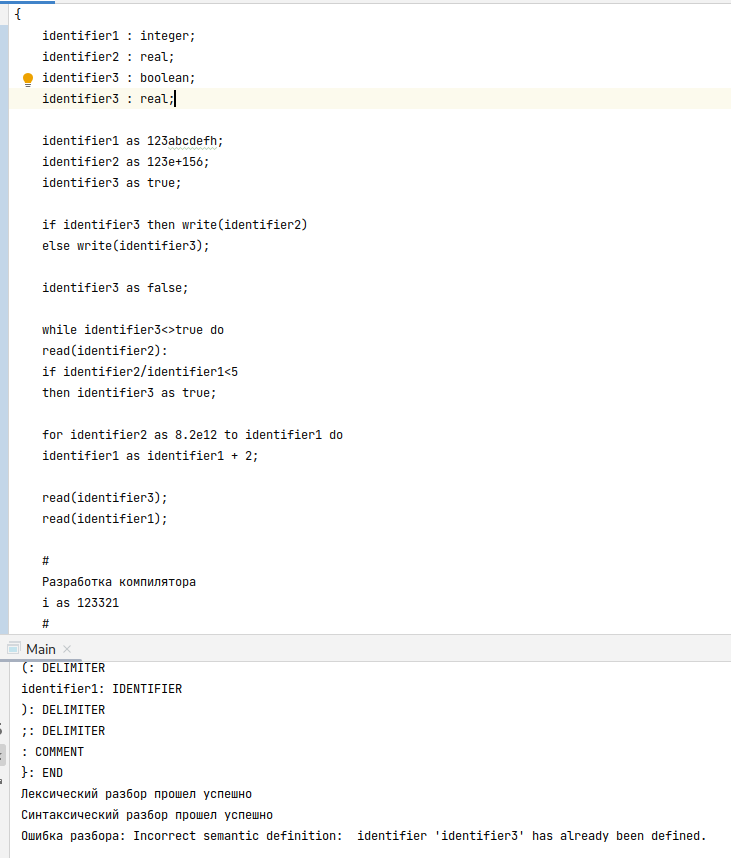


Рис.9. Ошибка: «Переменная уже существует»

4) Ошибка: «Нельзя применять операнды типа bool по отношению к переменным типа integer»

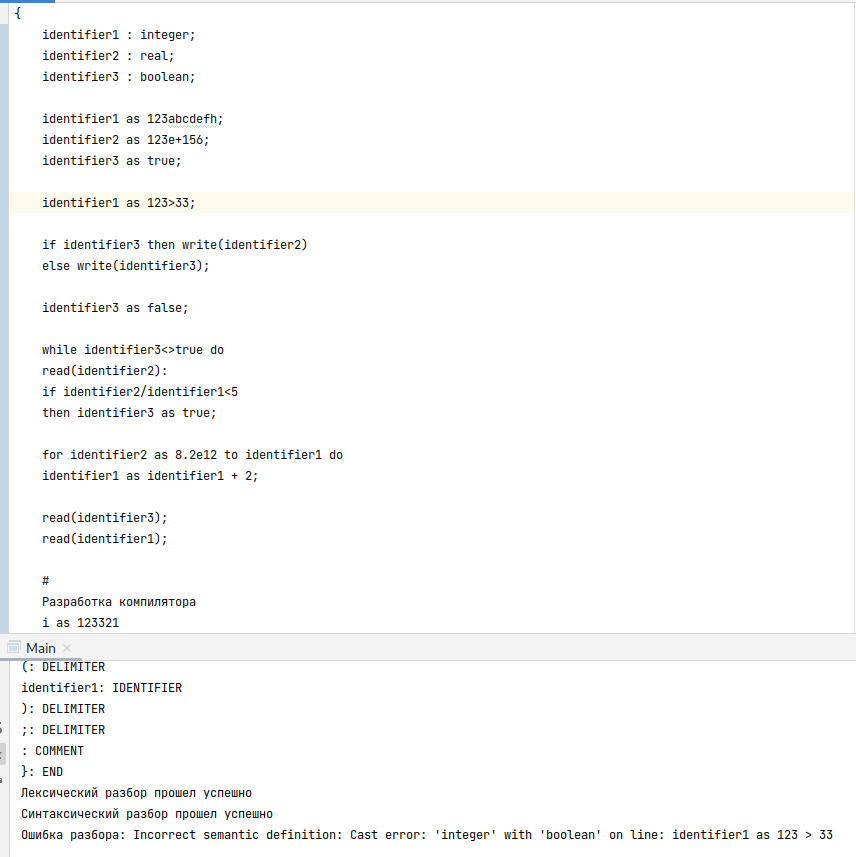


Рис.10. Ошибка: «Нельзя применять операнды типа bool по отношению к переменным int»

5) Ошибка: «Нельзя применять операнды численного типа по отношению к переменным типа boolean»

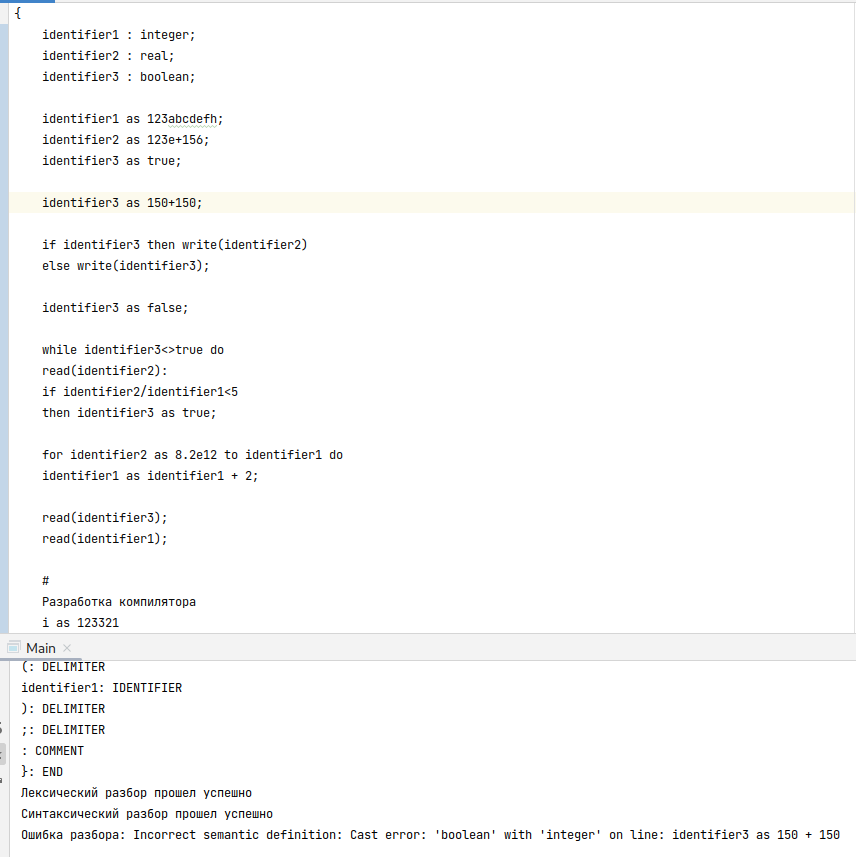


Рис.11. Ошибка: «Нельзя применять операнды численного типа по отношению к переменным типа bool»

6) Ошибка: «Оператор while требует наличие только boolean выражения в условии»

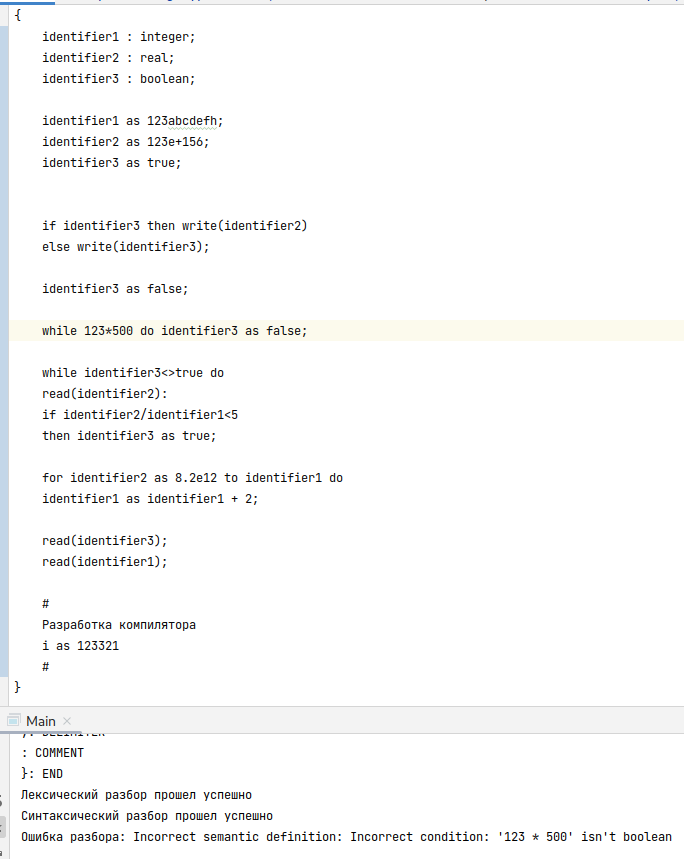


Рис.12. Ошибка: «Оператор while требует наличие только bool выражения в условии»

7) Ошибка: «Оператор for требует наличие оператора после ключевого слова do»

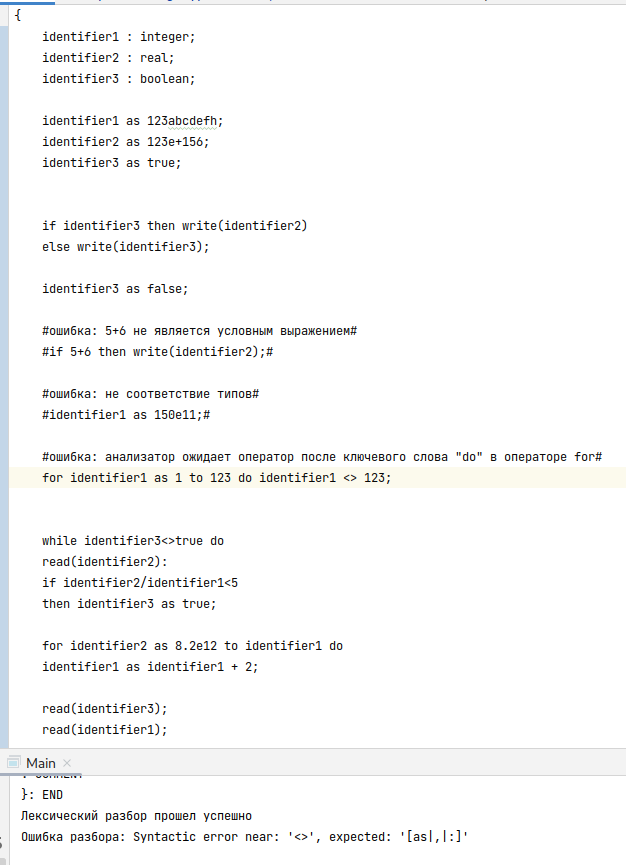


Рис.13. Ошибка: «Оператор for требует наличие оператора после ключевого слова do»

8) Ошибка: «Оператор условия требует наличие только boolean выражения»

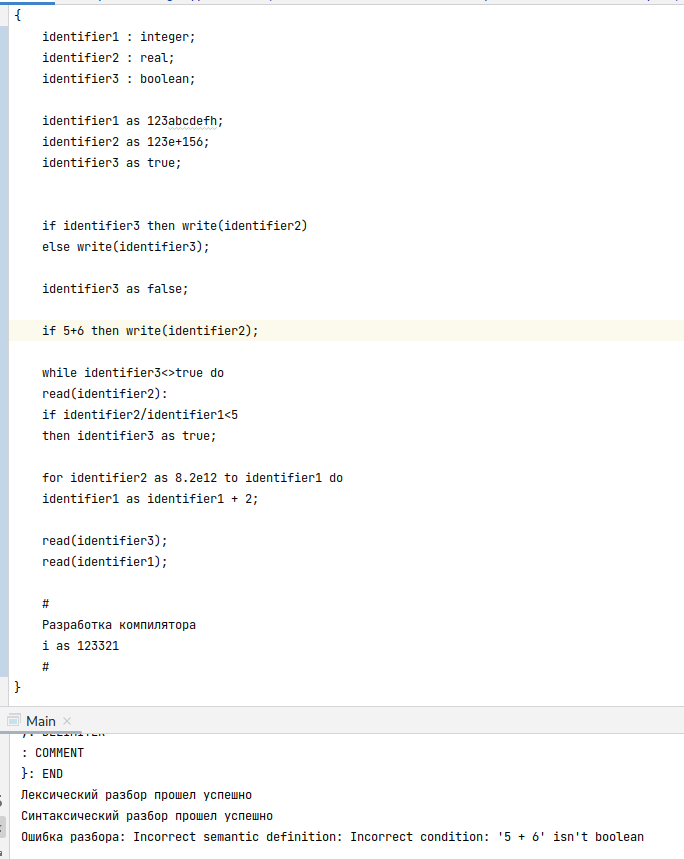


Рис.15. Ошибка: «Оператор условия требует наличие только bool выражения»

9) Ошибка: «Вы не можете назначить тип <тип1> к переменной типа <тип2>»

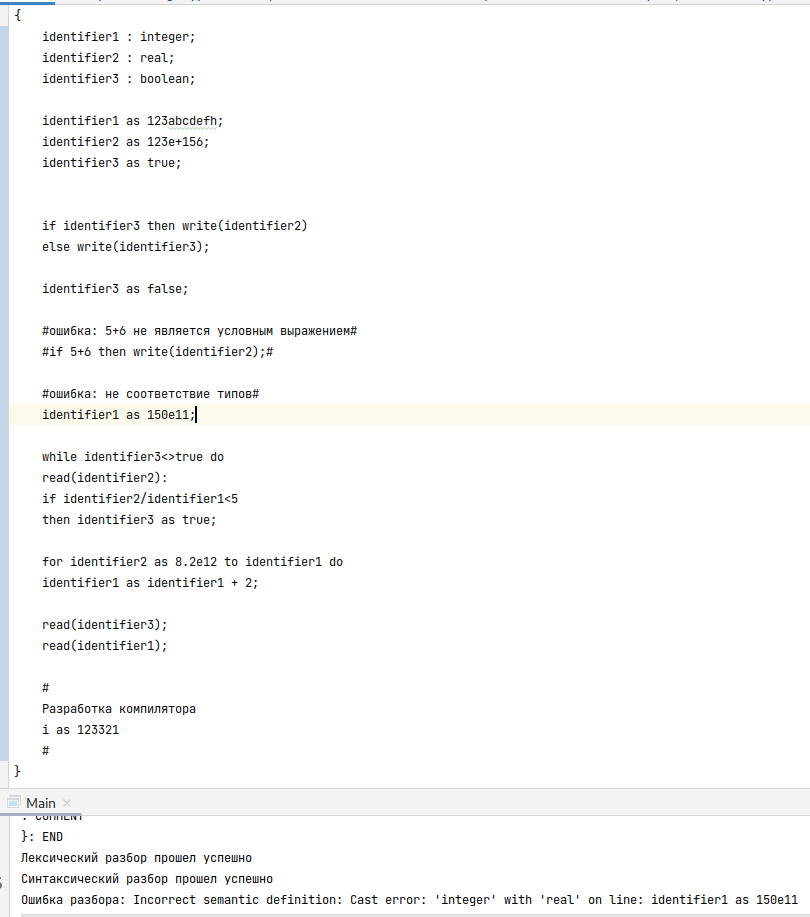


Рис.16. Ошибка: «Вы не можете назначить тип <тип1> к переменной типа <тип2>»

**Заключение**

В процессе работы была разработана часть компилятора модельного языка, который осуществляет лексический разбор формального языка. Компилятор реализован на языке Java.

Написано несколько содержательных примера программ, раскрывающих особенности конструкций учебного языка программирования, отразив в этих примерах все его функциональные возможности. Составлены таблицы лексем и диаграмма состояний с действиями для распознавания и формирования лексем языка.

Составлена блок схему лексического анализатора, для наглядного отображения процесса работы моей программы.

В процессе выполнения курсовой работы изучены методы лексического анализа формального M-языка, заданного по варианту.

**Список использованных источников**

1. Афанасьев А.Н. Формальные языки и грамматики: Учебное пособие. – Ульяновск: УлГТУ, 1997. – 84с. [текст]

2. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты.: Пер. с англ. – М.: Изд. дом «Вильямс», 2001. – 768с. [текст]

3. Братчиков И.Л. Синтаксис языков программирования / Под ред. С.С. Лаврова. – М.: Наука, 1975. - 262с. [текст]

4. Вайнгартен Ф. Трансляция языков программирования / Под ред. Мартынюка В.В.- М.: Мир, 1977. - 192с. [текст]

5. Вильямс А. Системное программирование в Windows 2000 для профессионалов. – СПб.: Питер, 2001. – 624с. [текст]

6. Волкова И.А., Руденко Т.В. Формальные языки и грамматики. Элементы теории трансляции. – М.: Диалог-МГУ, 1999. – 62с. [текст]

7. Грис Д. Конструирование компиляторов для цифровых вычислительных машин: Пер. с англ. – М.: Мир, 1975. – 544с. [текст]

8. Жаков В.И., Коровинский В.В., Фильчаков В.В. Синтаксический анализ и генерация кода. – СПб.: ГААП, 1993. – 26с. [текст]

9. Ишакова Е.Н. Теория формальных языков, грамматик и автоматов: Методические указания к лабораторному практикуму. – Оренбург: ГОУ ВПО ОГУ, 2004. – 54с. [текст]

10. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов. - М.: Мир, 1979. - 654с. [текст]

11. Рейуорд-Смит В. Теория формальных языков. Вводный курс: Пер. с англ. – М.: Радио и связь, 1988. – 128с. [текст]

12. Серебряков В.И. Лекции по конструированию компиляторов. – М.: МГУ, 1997. – 171с. [текст]

13. Соколов А.П. Системы программирования: теория, методы, алгоритмы: Учеб. пособие. – М.: Финансы и статистика, 2004. – 320с.

**Приложение А – Текст программы**

package ru;  
  
import ru.exception.ParseException;  
import ru.lexical.LexicalAnalyzer;  
import ru.util.TableUtil;  
import ru.semantic.SemanticAnalyzer;  
import ru.syntactical.SyntacticalAnalyzer;  
  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class Main {  
 public static void main(String[] args) {  
 *flushFiles*();  
 try {  
 new LexicalAnalyzer().analyze();  
 System.*out*.println("Лексический разбор прошел успешно");  
 new SyntacticalAnalyzer().analyze();  
 System.*out*.println("Синтаксический разбор прошел успешно");  
 new SemanticAnalyzer().analyze();  
 System.*out*.println("Семантический разбор прошел успешно");  
 }  
 catch (ParseException e){  
 System.*out*.println("Ошибка разбора: " + e.getMessage());  
 }  
  
 }  
  
 public static void flushFiles(){  
 String[] names = new String[]{  
 TableUtil.*lexTable*,  
 TableUtil.*tnPath*,  
 TableUtil.*tiPath*,  
 TableUtil.*lexemes*,  
 TableUtil.*descriptions*,  
 TableUtil.*expressions*,  
 TableUtil.*assignments*,  
 TableUtil.*conditionals*,  
 TableUtil.*tnbPath* };  
  
 for (String s: names){  
 File file = new File(s);  
 if(file.exists()){  
 try(FileWriter fw = new FileWriter(file)) {  
 fw.flush();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
 }  
 }  
}

package ru;  
  
public interface Analyzer {  
 void analyze();  
}  
package ru.exception.lex;  
  
import ru.exception.ParseException;  
  
public class IncorrectLexemeException extends ParseException {  
 private final String lexeme;  
 public IncorrectLexemeException(String lexeme){  
 super("Incorrect lexeme: " + lexeme);  
 this.lexeme = lexeme;  
 }  
 public String getLexeme() {  
 return lexeme;  
 }  
}  
package ru.exception.sem;  
  
import ru.exception.ParseException;  
  
public class SemanticException extends ParseException {  
 public SemanticException(String message) {  
 super("Incorrect semantic definition: " + message);  
 }  
}  
package ru.exception.syn;  
  
import ru.exception.ParseException;  
  
public class SyntacticException extends ParseException {  
  
 public SyntacticException(String lexeme, String expected) {  
 super("Syntactic error near: '" + lexeme + "', expected: '" + expected + "'");  
 }  
}

package ru.exception;  
  
public class ParseException extends RuntimeException{  
 public ParseException(String message){  
 super(message);  
 }  
}

package ru.lexical;  
  
import ru.Analyzer;  
import ru.exception.lex.IncorrectLexemeException;  
import ru.lexical.file.LexemeOuter;  
import ru.lexical.handler.NumberLexemeOuter;  
import ru.lexical.state.State;  
import ru.lexical.state.StateType;  
import ru.lexical.state.impl.CommentState;  
import ru.lexical.state.impl.IdentifierState;  
import ru.lexical.state.impl.delimiter.DelimiterState;  
import ru.lexical.state.impl.number.FloatState;  
import ru.lexical.state.impl.number.binary.BinaryState;  
import ru.lexical.state.impl.number.decimal.DecimalState;  
import ru.lexical.state.impl.number.oct.OctState;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class LexicalAnalyzer implements Analyzer {  
  
 private final List<State> states;  
  
 public LexicalAnalyzer() {  
 states = new ArrayList<>();  
 states.add(new IdentifierState());  
 states.add(new BinaryState());  
 states.add(new OctState());  
 states.add(new DecimalState());  
 states.add(new FloatState());  
 states.add(new CommentState());  
 states.add(new DelimiterState());  
  
 }  
 public void analyze() {  
 Reader reader = new Reader();  
 reader.setStateType(StateType.START);  
 while (!isNeedStopAnalyze(reader)) {  
 reader = readLexeme(reader);  
 if (reader.getStateType().equals(StateType.ERR)) {  
 throw new IncorrectLexemeException(reader.getBuffer());  
 }  
 if(!reader.getBuffer().equals("")) {  
 new LexemeOuter().out(reader.getBuffer());  
 if(isNumber(reader)){  
 new NumberLexemeOuter().handle(reader.getBuffer());  
 }  
 }  
 }  
 }  
  
 private Reader readLexeme(Reader reader){  
 reader.flush();  
 if (!Character.isWhitespace(reader.getCurrent())) {  
 reader = tryTransit(reader);  
 System.out.println(reader.getBuffer() + ": " + reader.getStateType());  
 } else {  
 reader.setStateType(StateType.WHITE\_SPACE);  
 reader.next();  
 }  
 return reader;  
 }  
  
 private Reader tryTransit(Reader reader) {  
 for (State s : states) {  
 if (s.is(reader.getCurrent())) {  
 return s.transit(reader);  
 }  
 }  
 reader.add();  
 reader.setStateType(StateType.ERR);  
 return reader;  
 }  
  
 private boolean isNeedStopAnalyze(Reader reader) {  
 return reader.getCurrent() == Character.MAX\_VALUE && !reader.charsExists();  
 }  
  
 private boolean isNumber(Reader reader){  
 StateType st = reader.getStateType();  
 return (st.equals(StateType.FLOAT)  
 || st.equals(StateType.DEC)  
 || st.equals(StateType.BINARY)  
 || st.equals(StateType.EXP)  
 || st.equals(StateType.DEC\_END)  
 || st.equals(StateType.FLOAT\_EXP)  
 || st.equals(StateType.OCT\_END)  
 || st.equals(StateType.OCT)  
 || st.equals(StateType.BINARY\_END)  
 || st.equals(StateType.HEX)  
 || st.equals(StateType.HEX\_END));  
 }  
}

package ru.lexical;  
  
import ru.lexical.file.FileLooker;  
import ru.util.TableUtil;  
import ru.lexical.state.StateType;  
  
import java.io.BufferedReader;  
import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
import java.nio.file.Paths;  
  
public final class Reader {  
 private final BufferedReader bufferedReader;  
 private char ch;  
 private String buffer;  
 private StateType stateType;  
 public Reader() {  
 try {  
 String codePath = "/home/almat/IdeaProjects/LexAnalyzer/src/res/code.txt";  
 bufferedReader = new BufferedReader(new FileReader(Paths.get(  
 codePath).toFile()));  
 } catch (FileNotFoundException e) {  
 throw new RuntimeException(e);  
 }  
 flush();  
 next();  
 }  
  
 public String getBuffer(){  
 return buffer;  
 }  
  
 public void flush(){  
 buffer = "";  
 }  
 public void add(){  
 buffer+=getCurrent();  
 }  
  
 public StateType getStateType() {  
 return stateType;  
 }  
  
 public void setStateType(StateType stateType) {  
 this.stateType = stateType;  
 }  
  
 public boolean charsExists() {  
 try {  
 return bufferedReader.ready();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
  
 public char getCurrent(){  
 return ch;  
 }  
  
 public void next(){  
 ch = gc();  
 }  
 private char gc(){  
 try {  
 int c = bufferedReader.read();  
 if(c == -1){  
 c = Character.MAX\_VALUE;  
 }  
 return (char) c;  
  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
 public boolean currentIsDelimiter() {  
 return new FileLooker(TableUtil.tlPath).look(String.valueOf(getCurrent())) != 0;  
 }  
  
}

package ru.lexical.file;  
import java.io.\*;  
import java.nio.file.Paths;  
import java.util.Arrays;  
import java.util.List;  
  
public class FileLooker {  
 private final String path;  
 public FileLooker(String path){  
 this.path = path;  
 }  
 public int look(String lexem){  
 int z;  
 try {  
 File file = new File(path);  
 if(!file.exists()){  
 file.createNewFile();  
 }  
 z = searchIndex(lines(), lexem);  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 return z;  
 }  
  
 private int searchIndex(List<String> lines, String lex){  
 return lines.indexOf(lex) +1;  
 /\*for (String s: lines){  
 if(s.equals(lex)){  
 return lines.indexOf(s)+1;  
 }  
 }  
 return 0;\*/  
 }  
 private List<String> lines() throws FileNotFoundException {  
 BufferedReader bufferedReader = new BufferedReader(  
 new FileReader(  
 Paths.get(path)  
 .toFile()));  
 return bufferedReader  
 .lines().toList();  
 }  
}  
package ru.lexical.file;  
  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class FileOuter {  
  
 private final String outPath;  
  
 public FileOuter(){  
 this.outPath =  
 "/home/almat/IdeaProjects/LexAnalyzer/" +  
 "src/res/out\_table/lex\_table.txt";  
 }  
  
 public void out(int t, int z){  
 try {  
 File file = new File(outPath);  
 if(!file.exists()){  
 file.createNewFile();  
 }  
 try (FileWriter fileWriter = new FileWriter(file,true)){  
 fileWriter.append(String.format("%d %d", t, z));  
 fileWriter.append("\n");  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
}  
package ru.lexical.file;  
  
import java.io.\*;  
  
public class FilePutter {  
  
 private final String path;  
  
 public FilePutter(String path){  
 this.path = path;  
 }  
  
  
 public int put(int t, String buffer){  
 try {  
 File file = new File(path);  
 if(!file.exists()){  
 file.createNewFile();  
 }  
 try(FileWriter fileWriter = new FileWriter(file,true)){  
 fileWriter.append(buffer);  
 fileWriter.append("\n");  
 }  
 return (int) new BufferedReader(new FileReader(file)).lines().count();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
}

package ru.lexical.file;  
import ru.util.TableUtil;  
import java.io.BufferedWriter;  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
  
public class LexemeOuter {  
 public void out(String lexeme) {  
 String path = TableUtil.lexemes;  
 File file = new File(path);  
 try {  
 BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file,true));  
 if(file.length() != 0){  
 bufferedWriter.newLine();  
 }  
 bufferedWriter.append(lexeme);  
 bufferedWriter.close();  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

package ru.lexical.handler;  
import ru.lexical.file.FileLooker;  
import ru.lexical.file.FileOuter;  
import ru.lexical.file.FilePutter;  
import ru.util.BinaryConverter;  
import ru.util.TableUtil;  
  
public class NumberLexemeOuter {  
  
 public void handle(String lexeme){  
 int z = new FileLooker(TableUtil.tnPath).look(lexeme);  
 if(z == 0) {  
 //lexeme = new BinaryConverter().binary(lexeme);  
 new FilePutter(TableUtil.tnbPath).put(TableUtil.tnNumber,  
 new BinaryConverter().binary(lexeme));  
 z = new FilePutter(TableUtil.tnPath)  
 .put(TableUtil.tnNumber, lexeme);  
 }  
 new FileOuter().out(TableUtil.tnNumber, z);  
 }  
  
}

package ru.lexical.handler;  
  
import ru.lexical.file.FileLooker;  
import ru.lexical.file.FileOuter;  
import ru.lexical.file.FilePutter;  
import ru.util.BinaryConverter;  
import ru.util.TableUtil;  
  
public class NumberLexemeOuter {  
  
 public void handle(String lexeme){  
 int z = new FileLooker(TableUtil.tnPath).look(lexeme);  
 if(z == 0) {  
 //lexeme = new BinaryConverter().binary(lexeme);  
 new FilePutter(TableUtil.tnbPath).put(TableUtil.tnNumber,  
 new BinaryConverter().binary(lexeme));  
 z = new FilePutter(TableUtil.tnPath)  
 .put(TableUtil.tnNumber, lexeme);  
 }  
 new FileOuter().out(TableUtil.tnNumber, z);  
 }  
  
}

package ru.lexical.handler;  
  
import ru.util.TypeConverter;  
  
public class NumberRepresentationConverter {  
 public String convert(String numberLexeme){  
 TypeConverter typeConverter = new TypeConverter();  
 String type = typeConverter.getType(numberLexeme);  
  
 if(type.equals("integer")){  
 return Integer.toBinaryString(Integer.parseInt(numberLexeme));  
 } else if (type.equals("real")) {  
 }  
 return null;  
 }  
}

package ru.lexical.state;  
public enum StateType {  
 START,  
 END,  
 ERR,  
 BINARY,  
 OCT,  
 DEC,  
 HEX,  
 BINARY\_END,  
 OCT\_END,  
 DEC\_END,  
 HEX\_END,  
 FLOAT,  
 FLOAT\_EXP,  
 EXP,  
 COMMENT,  
 DELIMITER,  
 IDENTIFIER,  
 WHITE\_SPACE,  
 KEY\_WORD,  
 BIGGER\_CHAR,  
 LESS\_CHAR  
}