



Федеральное государственное бюджетное образовательное учреждение
высшего образования "Московский технологический университет"
Направление 09.04.04 "Программная инженерия" (Магистр)

Отчет по домашней работе

по дисциплине "Разработка проблемно-ориентированных
транслирующих средств"

Выполнил:

Студент 1 курса
Заочного отделения
группы: ЗЖМЗ-01-16
Милев С. А.

Преподаватель:
Корягин С.В.

Москва, 2017 г.

Содержание

<u>Введение.....</u>	<u>3</u>
<u>1. Цель работы.....</u>	<u>4</u>
<u>2. Определение формы Бэкуса-Наура.....</u>	<u>5</u>
<u>3. Перечень ошибок.....</u>	<u>6</u>
<u>4. Схемы.....</u>	<u>8</u>
<u>5. Результаты работы программы.....</u>	<u>17</u>
<u>5.1. Растяжение пружины.....</u>	<u>17</u>
<u>5.2. Мутация клетки.....</u>	<u>18</u>
<u>5.3. Затухание светодиода.....</u>	<u>19</u>
<u>6. Примеры ошибок.....</u>	<u>20</u>
<u>Заключение.....</u>	<u>23</u>
<u>Список используемых источников.....</u>	<u>24</u>
<u>Приложение А. Листинг программы.....</u>	<u>25</u>

Введение

Целью работы является реализовать транслятор по БНФ форме.

Трансляция – преобразование программы, представленной на одном из языков программирования, в программу на другом языке и в определенном смысле равносильную первой.

Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов выдает для печати тексты программы и т.д.

Этапы трансляции:

1. *Лексический анализ* – процесс аналитического разбора входной последовательности символов (например, такой как исходный код на одном из языков программирования) с целью получения на выходе последовательности символов, называемых “токенами” (подобно группировке букв в словах). Группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется лексемой. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Токен можно представить в виде структуры содержащей идентификатор токена, если нужно, последовательности символов лексемы, выделенной из входного потока (строку, число и т.д.).

2. *Синтаксический анализ (парсинг)* – это процесс сопоставления последовательности лексем (слов, токенов) языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

3. Преобразование дерева разбора в другой язык программирования, либо интерпретирование дерева разбора.

1. Цель работы.

Разработать лексический и синтаксический анализатор по составленной форме Бэкуса-Наура. Анализатор должен отвечать следующим требованиям:

1. Анализ входного текста и определение его принадлежности к данной БНФ;
2. Определение места и типа первой ошибки во входном тексте
3. Расчёт выражений и вывод значений переменных.

2. Определение формы Бэкуса-Наура.

Цепочки языка могут содержать метасимволы, имеющие особое назначение. Метаязык, предложенный бэкусом и науром (бнф) использует следующие обозначения:

1. Символ « \Rightarrow » отделяет левую часть правила от правой (читается: «определяется как»);
2. Нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки « $\langle \rangle$ » и « \langle / \rangle »; форма
3. Терминалы - это символы, используемые в описываемом языке;
4. Правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты « $!$ » (читается: «или»).

Язык = "Begin;" Уравнения Состояния Коэффициенты Шаг
Промежуток Метод "End;"
Уравнения = "Expr:" Уравнение"," ... Уравнение ";"
Уравнение = ИмяУравнения "=" ПраваяЧасть
ИмяУравнения = "d" ИзвестнаяИмя
ПраваяЧасть = $\langle \text{Блок1} \text{ Знак3} \dots \text{Блок1} \rangle$
Знак3 = "+" ! "-"
Блок1 = Блок2 знак2... Блок2
Знак2 = "*" ! "/"
Блок2 = Блок3 "^" ... Блок3
Блок3 = ИзвестнаяИмя ! Число ! "(" ПраваяЧасть ")"
Состояния = "Vars0:" Коэффициент"," ... Коэффициент ";"
Коэффициенты = "Coeff:" Коэффициент"," ... Коэффициент ";"
Коэффициент = ИзвестнаяИмя "=" Число
Промежуток = "Range:" "[" Число ";" " Число "]" ";"
Шаг = "Step:" Число ";"
ИзвестнаяИмя = Буква \langle Символ...Символ \rangle
Символ = Буква ! Цифра
Метод = "Method:" ["euler" ! "runge-kut4"] ";"
Число = Целое ! Вещественное
Вещественное = Целое "." Целое
Целое = Цифра ... Цифра
Цифра = "0"! "1"! ... "9"
Буква = "A"! "B"! ... "Z"

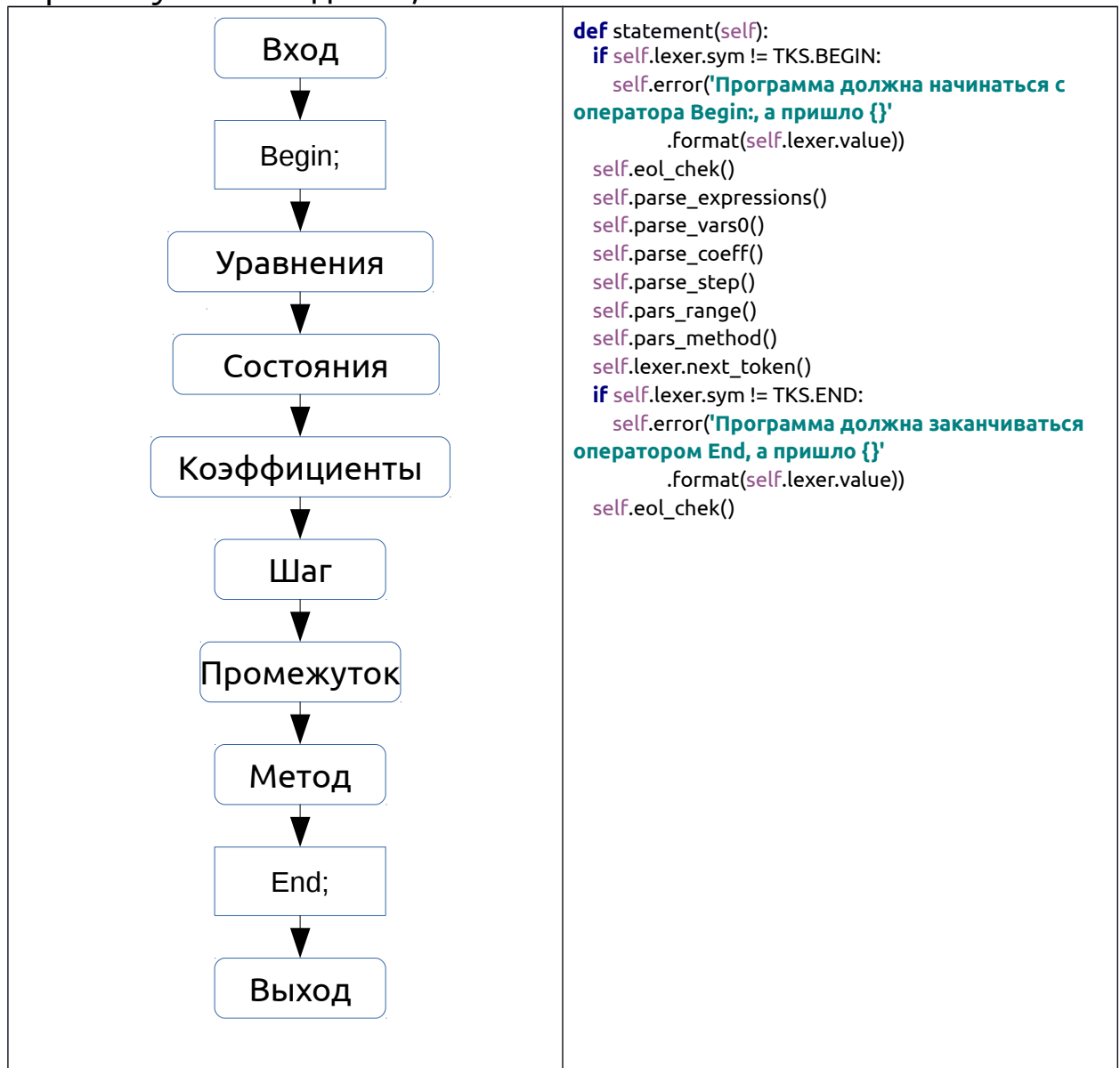
3. Перечень ошибок.

1. 'Отсутствует символ конца строки ";" в строке номер {}'
2. 'Непредвиденный символ "{}". Символ номер '
4. 'Непредвиденный символ "{}" при определении вещественного числа. Символ номер ',
5. 'Очень много точек в вещественном числе "{}". Символ номер ',
6. 'При обработки вещественного числа, пришел символ новой строки "\n". '
7. 'Скорее всего пропущен символ конца строки ";" Символ номер ',
8. '[Пропущен оператор определения интервала "Range". ',
9. 'Ожидался открывающий символ "[" при определении "Range", а пришел "{}". ',
10. 'Ожидался закрывающий символ "]" при определении "Range", а пришел "{}". ',
11. 'Непредвиденный символ "{}" при определении "Range". ',
12. 'Лишний символ "{}" при определении "Range". '
13. 'Пропущен оператор определения шага "Method". ',
14. 'Неизвестный метод интегрирования {}.'. '
15. 'Пропущен оператор определения шага "Step". '],
16. 'Пропущен оператор определения коэффициентов "Coeff". '],
17. 'Пропущен оператор определения начальных условий "Vars0". '],
18. 'Пропущен оператор определения дифференциальных уравнений "Expr". ',
19. 'Определение дифференциального уравнения должно начинаться с символа "d", а пришел "{}'. ',
20. 'Выражение "{}" не может начинаться со знака математической операции "{}'. ',
21. 'Ошибка в определении "{}". После математической функций "{}" должна идти скобка. ',
22. 'Ошибка определения "{}". Выражение не может заканчиваться знаком математической операции "{}'. ',
23. 'Непредвиденный символ "{}" в определении "{}'. ',
24. 'Два знака математических операций "{} {}" подряд в определении "{}'. ',
25. 'После числа может идти матоператор или ';)', а пришел '{}'. ',
26. 'После переменной может идти матоператор или ';)', а пришел '{}'. "
27. "Equal": 'Пропущен символ равенства "=" после переменной "{}" в операции "{}'. ',
28. 'Определено "{}" дифференциальных уравнений. Начальные условия предоставлены для "{}'. ',
29. 'Для "d{}" не определено начальное условие',

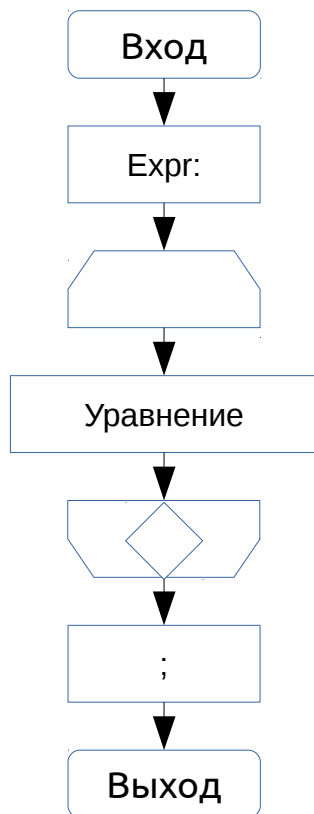
- 30. 'Имя уравнения "d{" совпадает с одним из коэффициентов "{",
- 31. "Ошибка в определении '{'. Неизвестная переменная '{. ",
- 32. 'Пропущен символ конца строки ";", в позиции '
- 33. 'Ошибка в определении имени переменной. '
- 34. 'Ошибка в определении значения переменной. '
- 35. 'Пропущен символ присваивания ":" оператора "{". '
- 36. 'Пропущенна закрывающая скобка при определении "{". '
- 37. 'Ошибка в определении "{". "(" больше чем ")"'
- 38. 'Ошибка в определении "{". "(" меньше чем ")"'
- 39. Программа должна начинаться с оператора Begin:, а пришло {'
- 40. 'Программа должна заканчиваться оператором End, а пришло {'
- 41. 'Неправильный синтаксис выражения'

4. Схемы

Язык = "Begin;" Уравнения Состояния Коэффициенты Шаг
Промежуток Метод "End;"



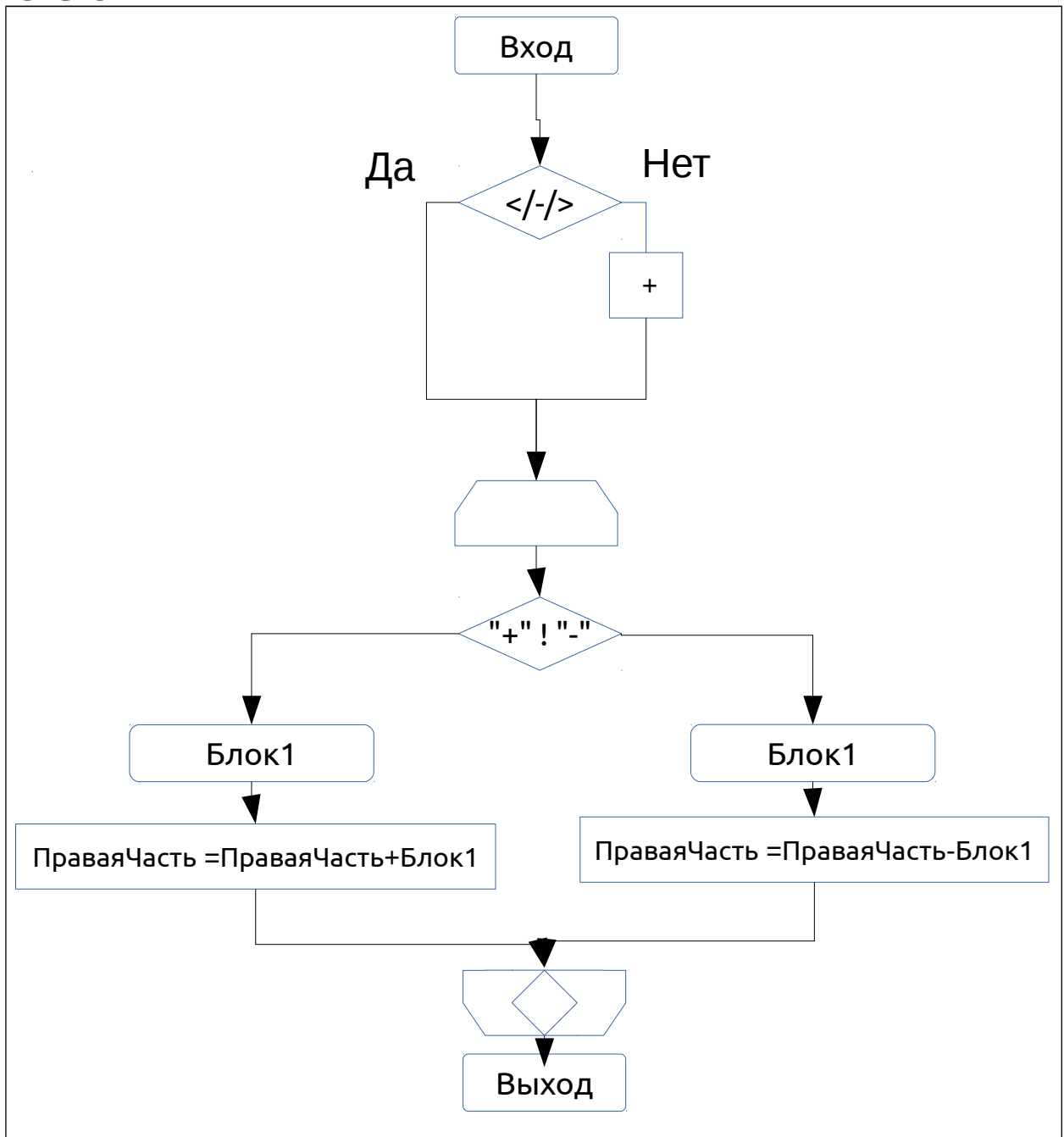
Уравнения = "Expr:" Уравнение", " ... Уравнение";"



```

def parse_expressions(self):
    self.lexer.next_token()
    op = self.lexer.value
    if self.lexer.sym != TKS.EXPR:
        self.error(ERR.PARS_ERRORS["Expr"][0])
    self.colon_check(op)
    expr_dict = {}
    while self.lexer.ch != ';':
        k = self.check_diffname()
        self.check_equ(k, op)
        v = self.parse_expression(k)
        if v.count('(') > v.count(')'): self.error(
            'Ошибка в определении "{}". "(" больше чем
            ")"'.format(k))
        elif v.count('(') < v.count(')'): self.error(
            'Ошибка в определении "{}". "(" меньше чем
            ")"'.format(k))
        expr_dict[k] = v
        if self.lexer.ch == ';':
            self.lexer.next_token()
            # self.lexer.next_token()
        else:
            break
    self.eol_chek()
    self.node.add_too_tree('Expr', expr_dict)
  
```

ПраваяЧасть = </-/>Блок1 ЗнакЗ... Блок1
ЗнакЗ = "+" ! "-"



```

def parse_expression(self, key):
    value = ""
    if self.lexer.ch in TKS.MATHOP: self.error(ERR.PARS_ERRORS["Expr"][2].format(key, self.lexer.ch))
    while self.lexer.ch not in ',,:':
        self.lexer.next_token()
        if self.lexer.sym == TKS.NUM:
            if self.lexer.ch in TKS.MATHOP or self.lexer.ch in ',,:':
                value += self.lexer.value
            else:
                self.error(ERR.PARS_ERRORS["Expr"][7].format(self.lexer.ch))
        elif self.lexer.sym == TKS.VAR:
            if self.lexer.ch in TKS.MATHOP or self.lexer.ch in ',,:':
                value += self.lexer.value
  
```

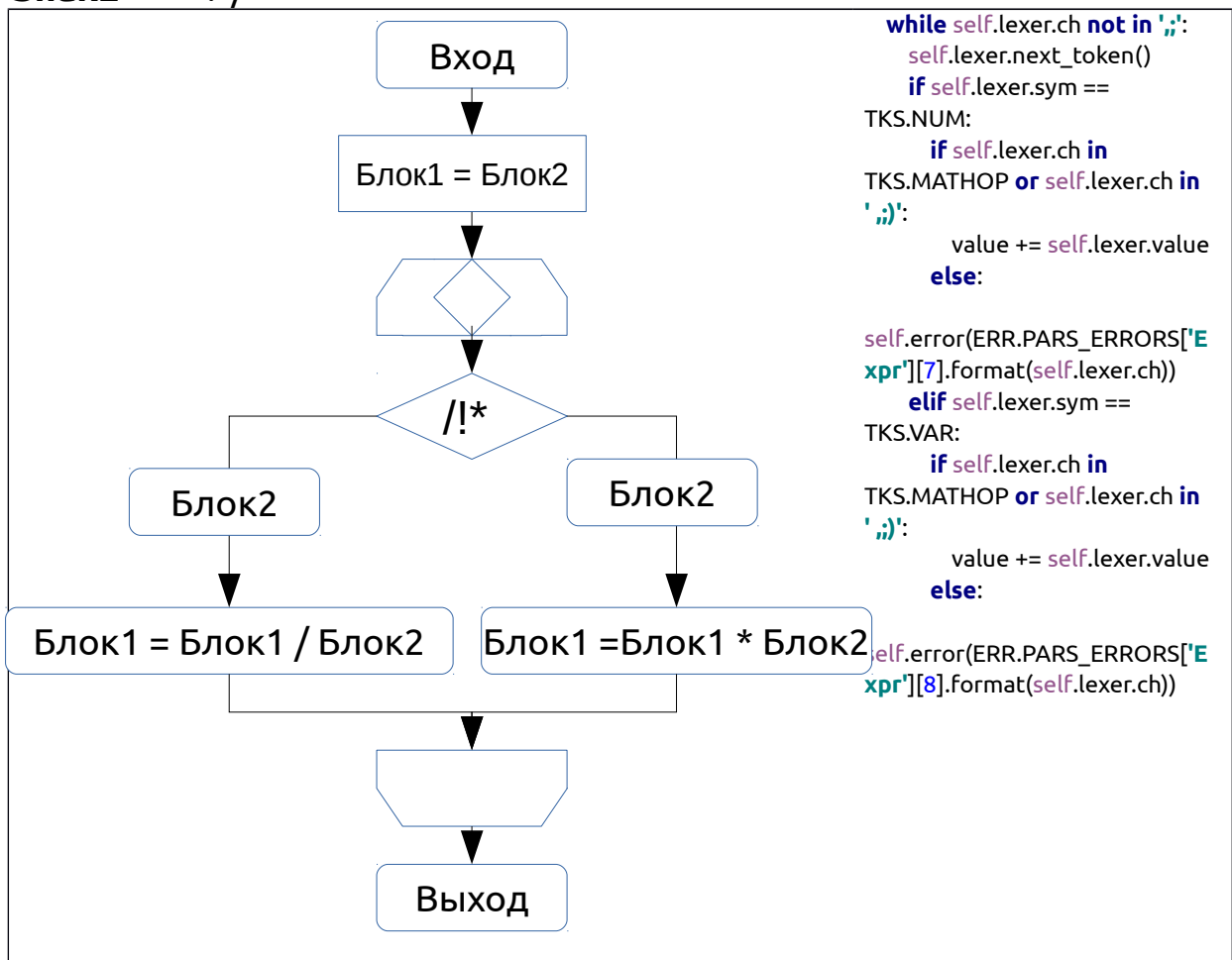
```

else:
    self.error(ERR.PARS_ERRORS['Expr'][8].format(self.lexer.ch))
elif self.lexer.sym == TKS.RPAR:
    value += self.lexer.value
elif self.lexer.sym == TKS.LPAR:
    value += self.lexer.value + self.rpar_chek(key)
elif self.lexer.value in TKS.MATHFUNC:
    if self.lexer.ch != '(':
        self.error(ERR.PARS_ERRORS['Expr'][3].format(key, self.lexer.value))
    else:
        value += self.lexer.value + self.rpar_chek(key)
elif self.lexer.value in TKS.MATHOP:
    if self.lexer.ch in TKS.MATHOP:
        self.error(ERR.PARS_ERRORS['Expr'][6].format(self.lexer.value, self.lexer.ch, key))
    else:
        value += self.lexer.value
else:
    self.error(ERR.PARS_ERRORS['Expr'][5].format(self.lexer.value, key))
if value[-1] in TKS.MATHOP:
    self.error(ERR.PARS_ERRORS['Expr'][4].format(key, value[-1]))
return value

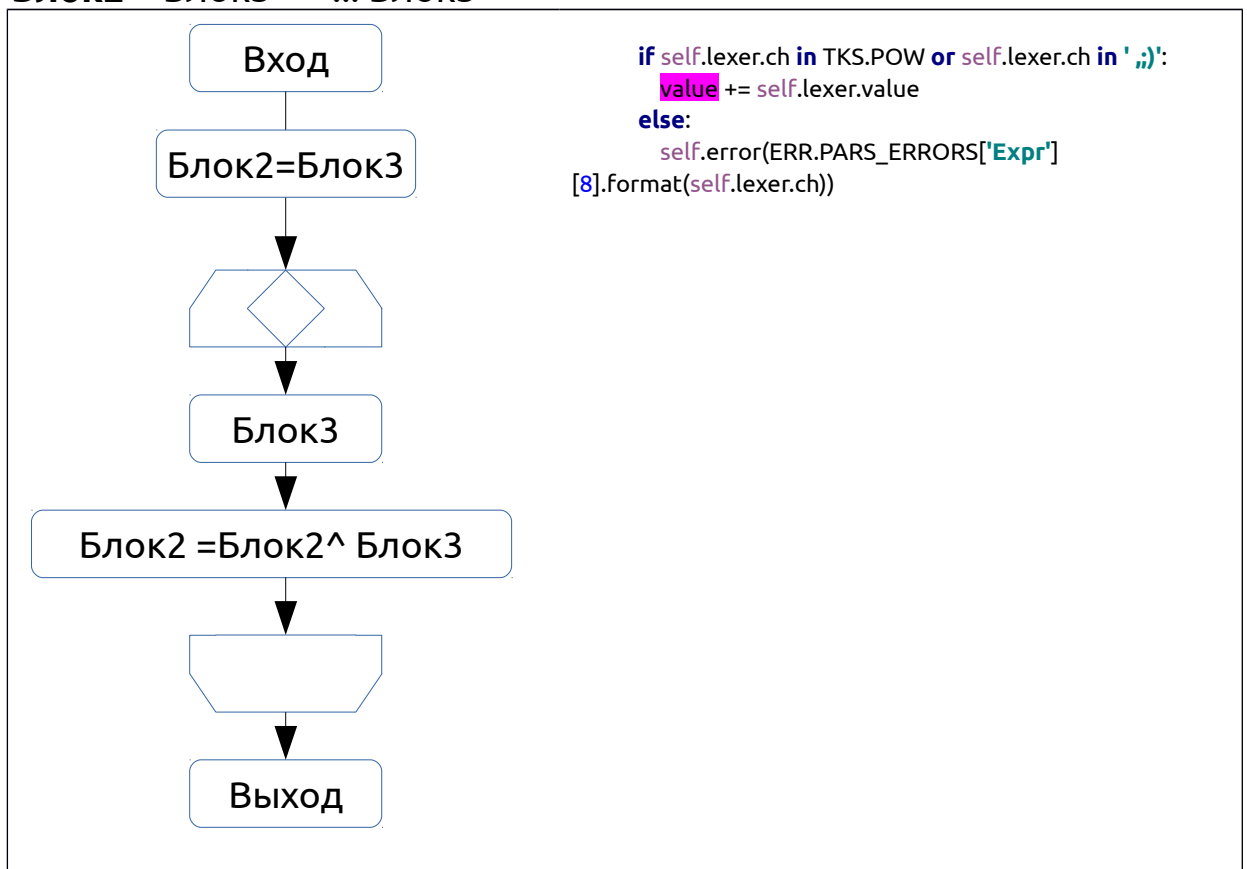
```

Блок1 = Блок2 знак2... Блок2

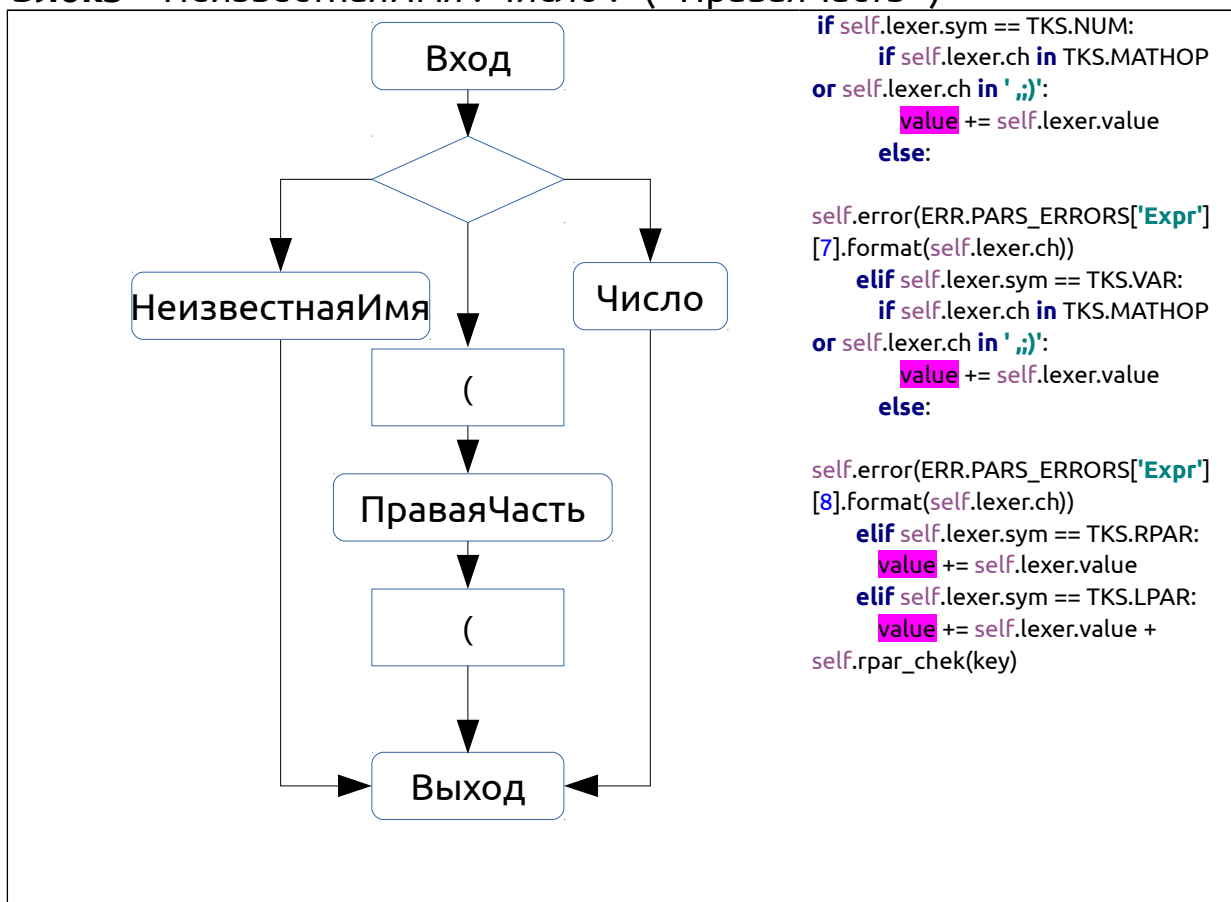
Знак2 = "*"!" /"



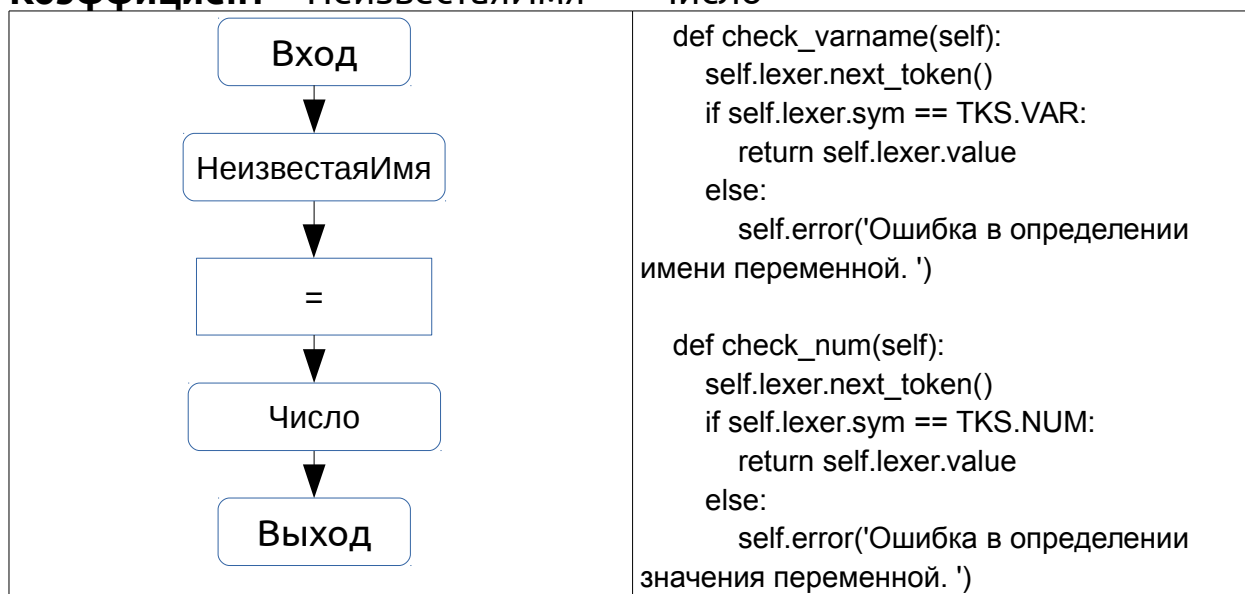
Блок2 = Блок3 "^" ... Блок3



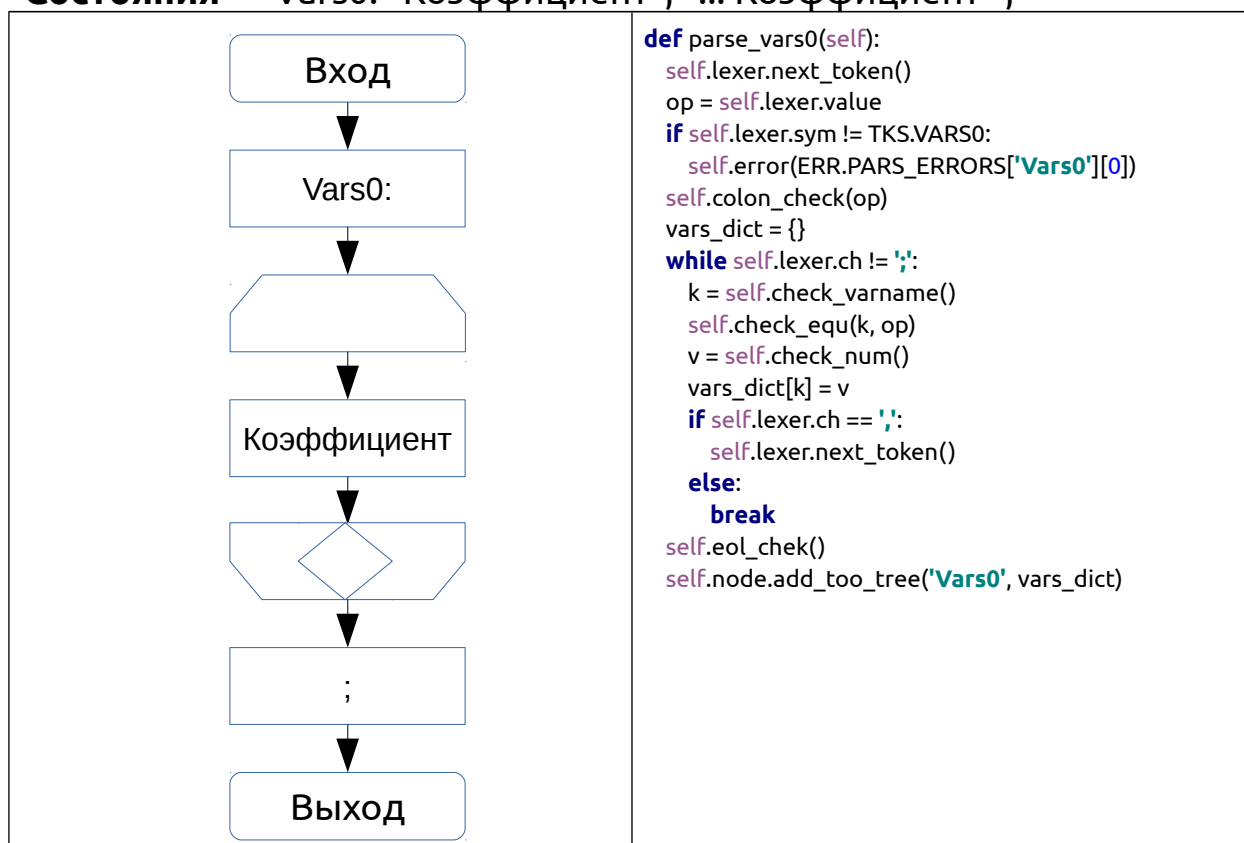
Блок3 = НеизвестнаяИмя ! Число ! "(" ПраваяЧасть ")"



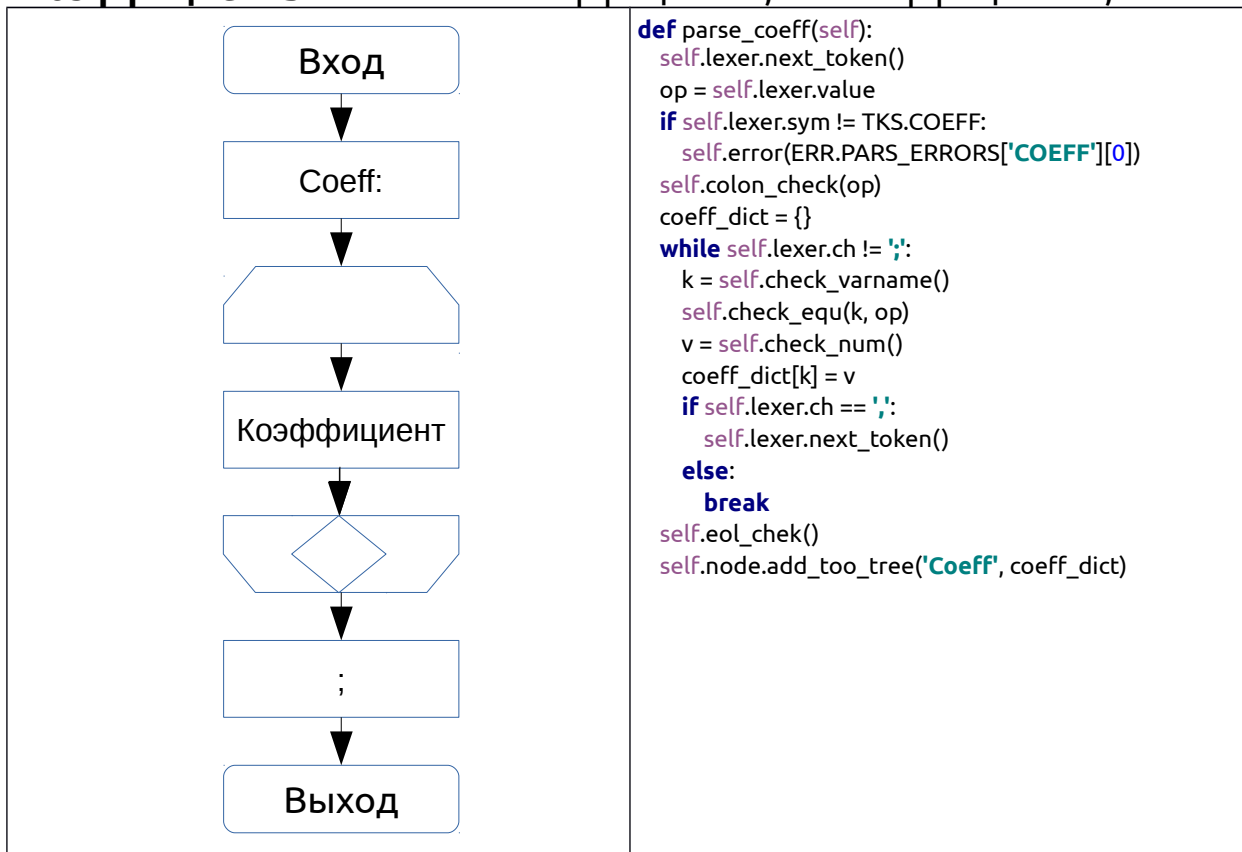
Коэффициент = НеизвестаяИмя "=" Число



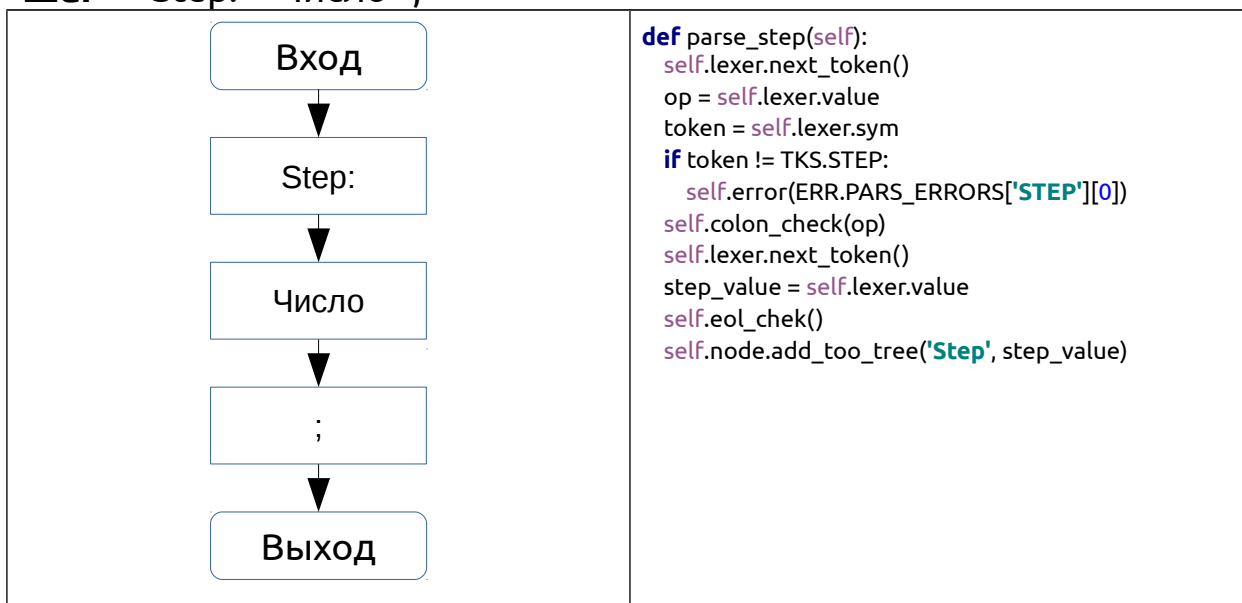
Состояния = "Vars0:" Коэффициент", " ... Коэффициент ";"



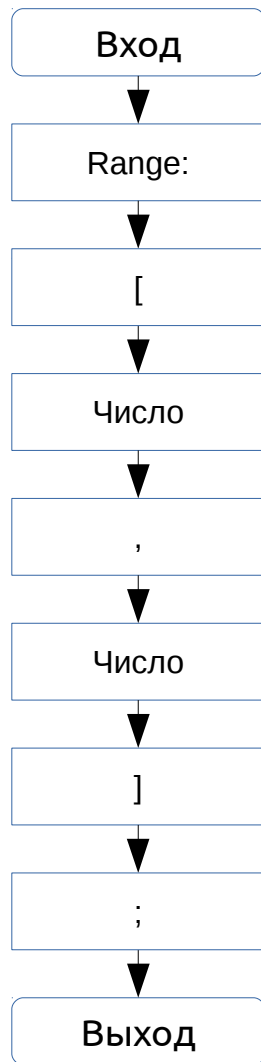
Коэффициенты = "Coeff:" Коэффициент", " ... Коэффициент ";"



Шаг = "Step: " Число ";"



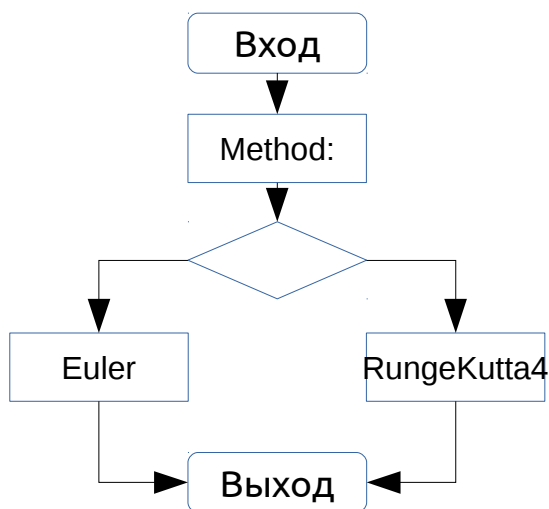
Промежуток = "Range:" "[" Число ";" " Число "]" ";"



```

def pars_range(self):
    self.lexer.next_token()
    op = self.lexer.value
    token = self.lexer.sym
    if token != TKS.RANGE:
        self.error(ERR.PARS_ERRORS["RANGE"][0])
    self.colon_check(op)
    self.lexer.next_token()
    if self.lexer.sym != TKS.LSB:
        self.error(ERR.PARS_ERRORS["RANGE"][1]
                    .format(self.lexer.value))
    else:
        range_value = self.lexer.value
        while self.lexer.sym != TKS.RSB:
            if self.lexer.sym == TKS.SEMICOLON:
                self.error(ERR.PARS_ERRORS["RANGE"]
                            [2].format(self.lexer.value))
            elif self.lexer.sym == TKS.NUM or self.lexer.value
                in '[':
                self.lexer.next_token()
            else:
                self.error(ERR.PARS_ERRORS["RANGE"]
                            [3].format(self.lexer.value))
            range_value += self.lexer.value
        for l in '[':
            if range_value.count(l) > 1:
                self.error(ERR.PARS_ERRORS["RANGE"][4].format(l))
        self.node.add_too_tree('Range', range_value) # Add
        too tree
        self.eol_chek()
  
```

Метод = "Method:" ["euler" ! "runge-kut4"] ";"

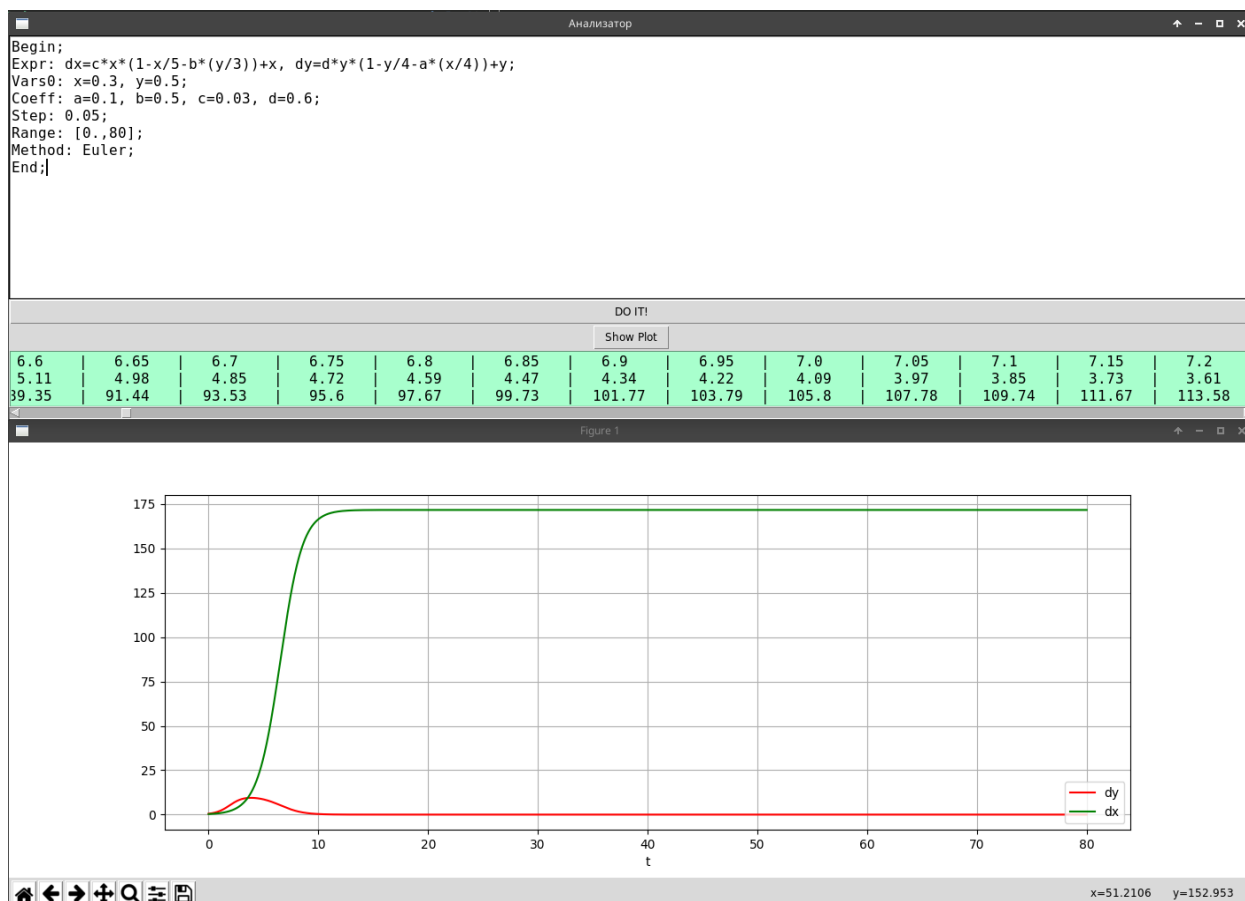


```

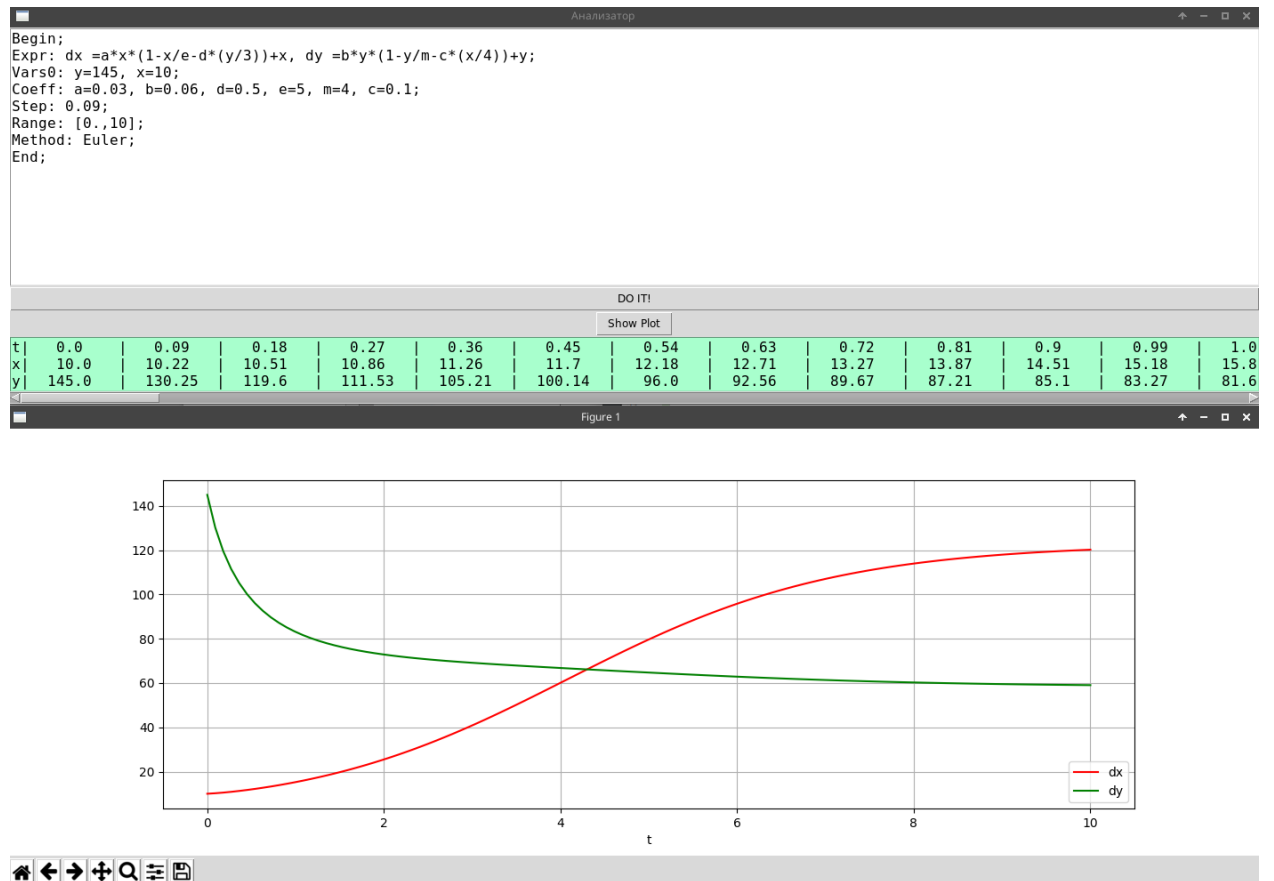
def pars_method(self):
    self.lexer.next_token()
    op = self.lexer.value
    if self.lexer.sym != TKS.METHOD:
        self.error(ERR.PARS_ERRORS["METHOD"][0])
    self.colon_check(op)
    self.lexer.next_token()
    token = self.lexer.value
    if self.lexer.value not in TKS.METHODS:
        self.error(ERR.PARS_ERRORS["METHOD"]
                    [1].format(self.lexer.value))
    method_value = self.lexer.value
    self.eol_chek()
    self.node.add_too_tree('Method', method_value)
  
```


5. Результаты работы программы

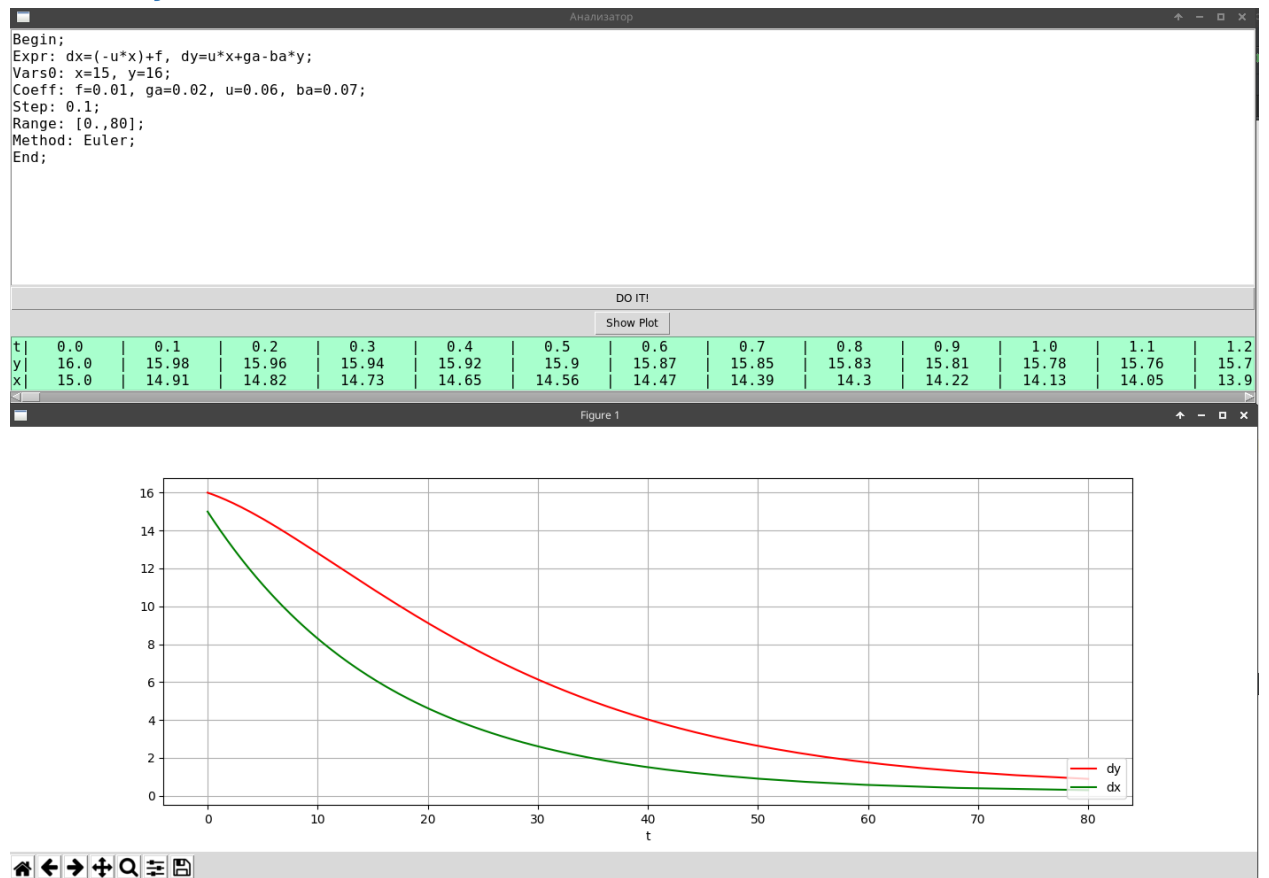
5.1. Растяжение пружины



5.2. Мутация клетки

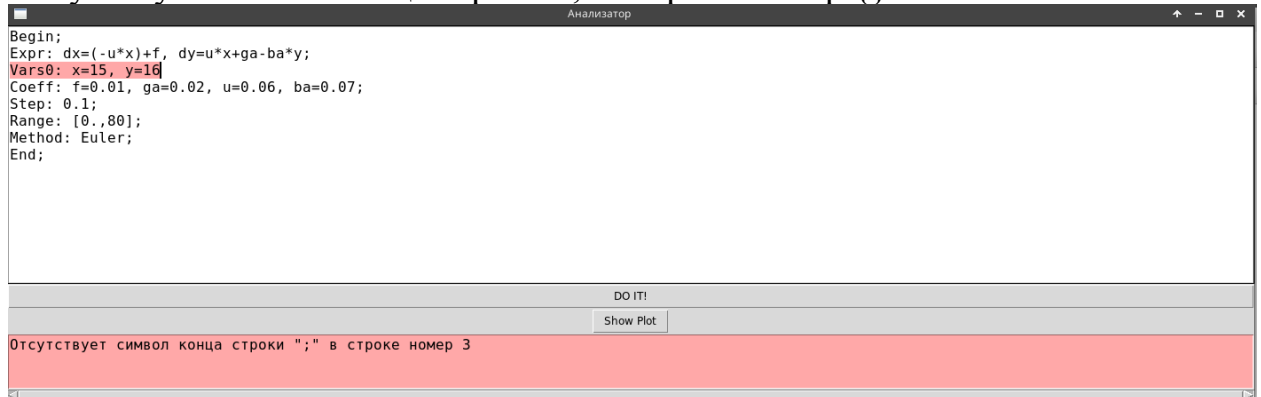


5.3. Затухание светодиода



6. Примеры ошибок

Отсутствует символ конца строки ";" в строке номер {}



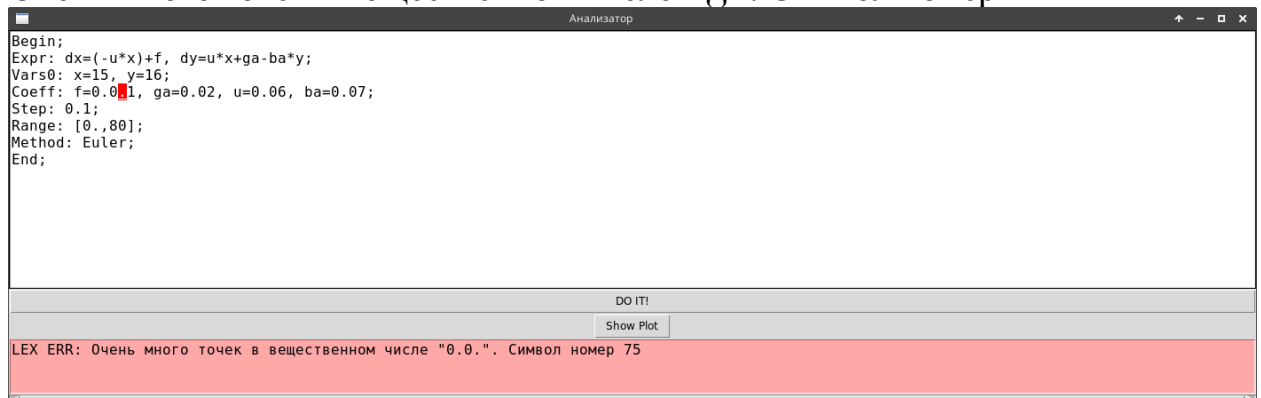
```
Begin;
Expr: dx=(-u*x)+f, dy=u*x+ga-ba*y;
Vars0: x=15, y=16;
Coeff: f=0.01, ga=0.02, u=0.06, ba=0.07;
Step: 0.1;
Range: [0.,80];
Method: Euler;
End;
```

DO IT!

Show Plot

Отсутствует символ конца строки ";" в строке номер 3

Очень много точек в вещественном числе "{}". Символ номер '



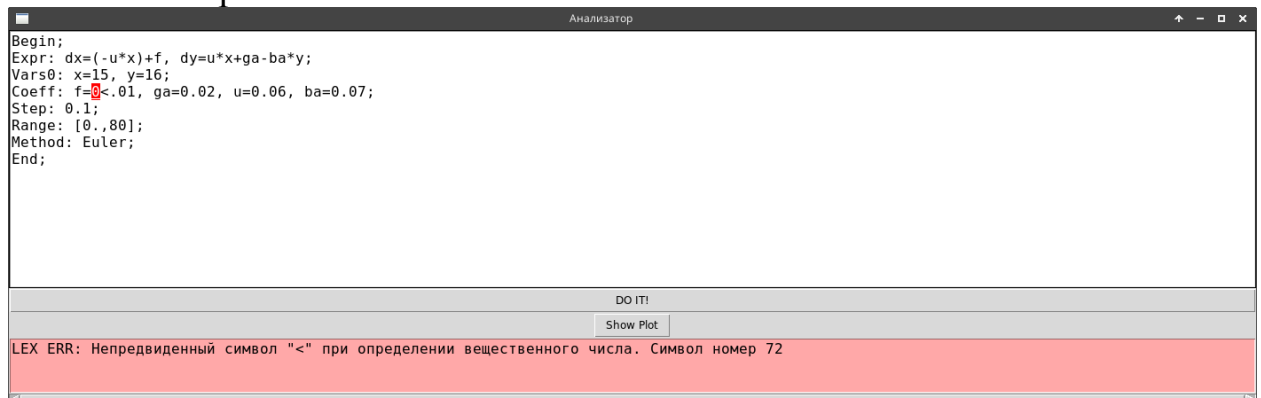
```
Begin;
Expr: dx=(-u*x)+f, dy=u*x+ga-ba*y;
Vars0: x=15, y=16;
Coeff: f=0.001, ga=0.02, u=0.06, ba=0.07;
Step: 0.1;
Range: [0.,80];
Method: Euler;
End;
```

DO IT!

Show Plot

LEX ERR: Очень много точек в вещественном числе "0.0.". Символ номер 75

'Непредвиденный символ "{}" при определении вещественного числа. Символ номер '



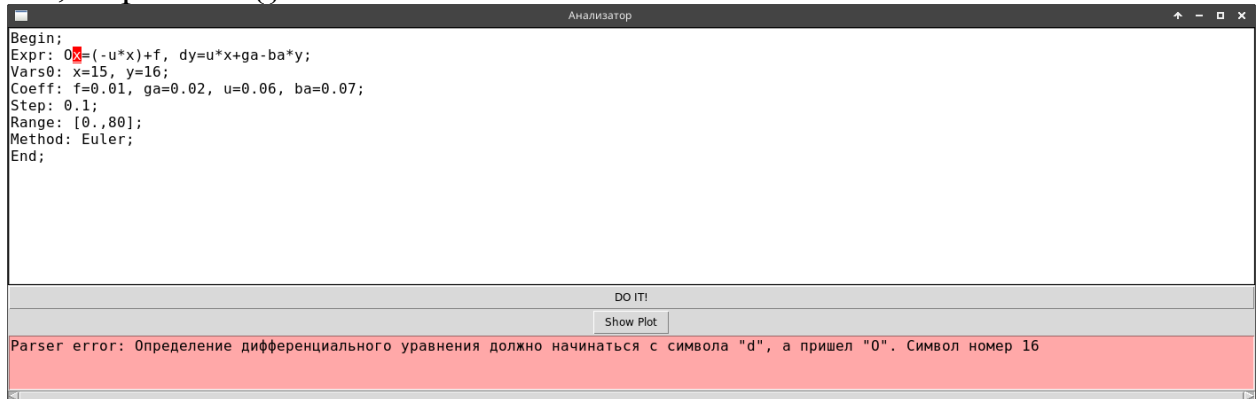
```
Begin;
Expr: dx=(-u*x)+f, dy=u*x+ga-ba*y;
Vars0: x=15, y=16;
Coeff: f=<.01, ga=0.02, u=0.06, ba=0.07;
Step: 0.1;
Range: [0.,80];
Method: Euler;
End;
```

DO IT!

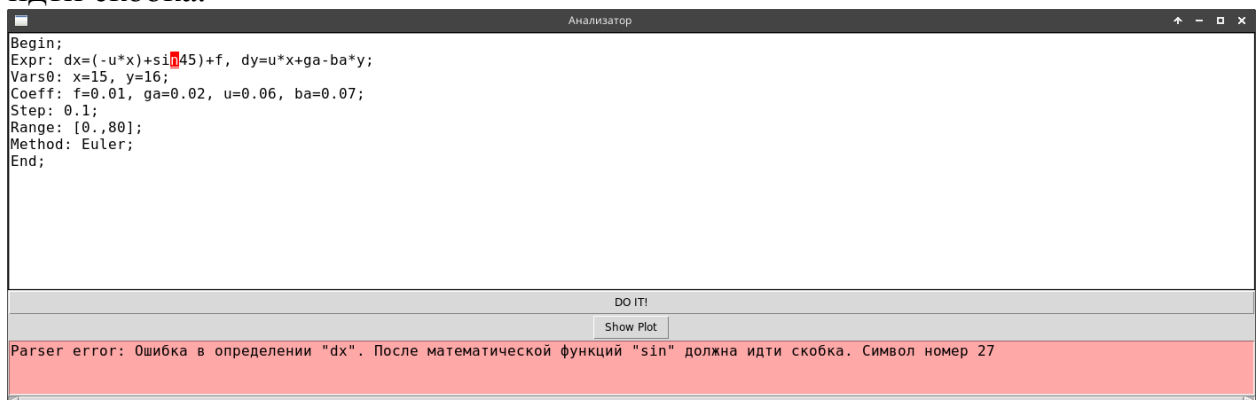
Show Plot

LEX ERR: Непредвиденный символ "<" при определении вещественного числа. Символ номер 72

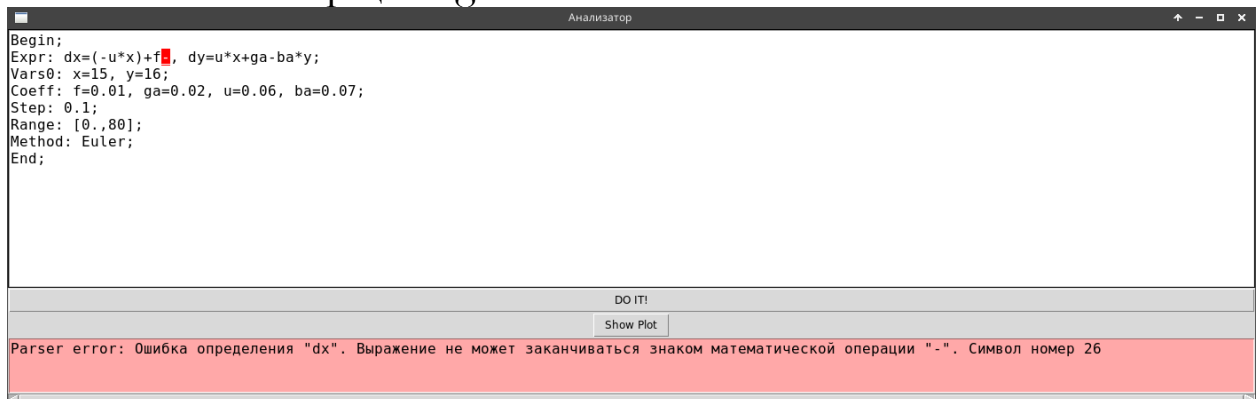
'Определение дифференциального уравнения должно начинаться с символа "d", а пришел "{}'.



'Ошибка в определении "{}". После математической функций "{}" должна идти скобка.



'Ошибка определения "{}". Выражение не может заканчиваться знаком математической операции "{}'.



Два знака математических операций "{}" подряд в определении "{}". '

```
Begin;  
Expr: dx=(-u*x)+f-, dy=u*x+ga-ba*y;  
Vars0: x=15, y=16;  
Coeff: f=0.01, ga=0.02, u=0.06, ba=0.07;  
Step: 0.1;  
Range: [0.,80];  
Method: Euler;  
End;
```

DO IT!

Show Plot

Parser error: Два знака математических операций "+" подряд в определении "dx". Символ номер 24

'Определено "{}" дифференциальных уравнений. Начальные условия предоставлены для "{}".'

```
Begin;  
Expr: dx=(-u*x)+f, dy=u*x+ga-ba*y;  
Vars0: y=16;  
Coeff: f=0.01, ga=0.02, u=0.06, ba=0.07;  
Step: 0.1;  
Range: [0.,80];  
Method: Euler;  
End;
```

DO IT!

Show Plot

Ошибка постобработчика: Определено "2" дифференциальных уравнений. Начальные условия предоставлены для "1".

"Ошибка в определении '{}'. Неизвестная переменная '{}". "

```
Begin;  
Expr: dx=(-u*x)+s+f, dy=u*x+ga-ba*y;  
Vars0: x=15, y=16;  
Coeff: f=0.01, ga=0.02, u=0.06, ba=0.07;  
Step: 0.1;  
Range: [0.,80];  
Method: Euler;  
End;
```

DO IT!

Show Plot

Ошибка постобработчика: Ошибка в определении 'dx'. Неизвестная переменная 's'.

Заключение

В данной работе была поставлена задача разработки синтаксически управляемого транслятора по БНФ форме.

В ходе решения задачи была разработана программа с графическим интерфейсом. Основные классы программы:

1. **PreLexer** – класс реализует предварительный анализ. Проверяет синтаксическую целостность и корректную разметку полученных данных.
2. **Lexer** – класс производит лексический анализ введенного текста, производит формирование узлов, для последующего синтаксического анализа.
3. **Parser** – класс производит синтаксический анализ, отвесает за корректную последовательность введенных данных. Производит формирование конечного лексического дерева принимаемого классами, отвечающими за математическую обработку.
4. **PostParserHandler** — класс отвечает за конечную проверку лексического дерева, проверяет целостность данных и их корректную последовательность.
5. **MyUI** — класс реализует графический интерфейс для ввода и вывода данных.

Программа производит анализ исходного текста и выдает сообщения об ошибках, если они есть. Задача была реализована на языке Python3 с применением IDE Pycharm. Графический интерфейс реализован средствами пакета Tkinter, входящем в стандартную библиотеку Python3. В ходе тестирования реализованного программного обеспечения было выявлено, что программа решает задачу корректно и устойчиво работает на тестовом наборе данных.

Список используемых источников.

1. Кнут Д. Искусство программирования, том 3. Сортировка и поиск – 2-е изд. – М.: «Вильямс», 2007. -824 с.
2. Эккель Б. Философия C++. Практическое программирование. – М.: «Питер», 2004. – 608 с.
3. Макконелл, Дж. Основы современных алгоритмов. 2-е дополнительное издание – М.: Техносфера, 2004. – 368 с.
4. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.Ж Мир, 1979. – 384 с
5. Никлаус Вирт. Построение компиляторов – М.: «ДМК Пресс», 2010. – 192 с.

Приложение А. Листинг программы.

```
==> TOKENS.py <==
COMMA, RSB, LSB, NUM, STEP, METHOD, COLON, SEMICOLON, \
PLUS, MINUS, EOF, DOT, VAR, BEGIN, END, RANGE, LPAR, RPAR, \
RANGEVAL, EULER, RUNGKUT2, RUNGKUT4, COEFF, EQUAL, VARS0, POW, \
MULT, DIVIS, SIN, COS, TANG, EXPR = range(32)
WORDS = {'Step': STEP,
         'Method': METHOD,
         'Begin': BEGIN,
         'Range': RANGE,
         'End': END,
         'Coeff': COEFF,
         'Vars0': VARS0,
         'Expr': EXPR,
        }
SYMBOLS = {';': SEMICOLON,
          '=': EQUAL,
          ':': COLON,
          ',': COMMA,
         }
BRACKETS = {'[': LSB,
            ']': RSB,
           }
MATHOP = {'/': DIVIS,
          '+': PLUS,
          '*': MULT,
          '-': MINUS,
          '^': POW,
         }
MATHFUNC = {'sin': SIN,
            'cos': COS,
            'tg': TANG,
           }
PARANTH = {'(': LPAR,
           ')': RPAR,
          }
DELIMETORS = {'.': DOT}
METHODS = {'Euler': EULER,
           'RungeKutta2': RUNGKUT2,
           'RungeKutta4': RUNGKUT4,
          }
```

```

==> lexex.py <==
import sys
from celery.apps.multi import Node
import TOKENS as TKS
import ERRORS as ERR
class PreLexer:
    def __init__(self, file):
        plf = open(file)
        self.all_lines_arr = [ls.rstrip() for ls in plf]
        plf.close()
        self.chek_eols()
    def error(self, msg):
        f = open('pre_lex_err.txt', 'w')
        print(msg)
        f.write(msg)
        f.close()
        sys.exit(1)
    def chek_eols(self):
        for k, v in enumerate(self.all_lines_arr):
            if not v.endswith(';'):
                self.error(ERR.PRE_LEX_ERRORS[0].format(k+1))
class Lexer:
    """
    Parsing Char by chr and returning tokens
    """
    def __init__(self, file):
        self.raw_text_file = open(file)
        self.ch = ' '
    def error(self, msg):
        char_no = self.raw_text_file.tell()
        print('LEX ERR: ' + msg + str(char_no))
        self.raw_text_file.close()
        f = open('lex_err.txt', 'w')
        LEX_ERROR = 'LEX ERR: ' + msg + str(char_no)
        f.write(LEX_ERROR)
        f.close()
        sys.exit(1)
    def lex_comma(self):
        """
        NOT USSING YET. LET IT BE HERE
        :return:
        """
        if self.ch != ',':
            self.error('Ожидался символ ",", а пришел {}'.format(self.ch))
        else:
            return ','
    def lex_num(self):
        num = ""
        while self.ch not in [l for k in [TKS.SYMBOLS, TKS.BRACKETS, TKS.MATHOP,
TKS.PARANTH] for l in k]:
            if self.ch.isdigit() or self.ch in TKS.DELIMETORS:
                num += self.ch
                self.getc()
                if num.count('.') > 1:
                    self.error(ERR.LEX_ERRORS[2].format(num))
            elif self.ch == "\n":

```

```

        self.error(ERR.LEX_ERRORS[3].format(num))
    else:
        self.error(ERR.LEX_ERRORS[1].format(self.ch))
    if num[-1] == '.': num += '0'
    return num
def getc(self):
    self.ch = self.raw_text_file.read(1)
def next_token(self):
    self.value = None
    self.sym = None
    while self.sym == None:
        if len(self.ch) == 0:
            self.sym = TKS.EOF
            self.raw_text_file.close()
        # elif not self.ch in '\n\t':
        #     self.getc()
        elif self.ch.isspace():
            self.getc()
        elif self.ch in TKS.MATHOP:
            self.sym = TKS.MATHOP[self.ch]
            self.value = self.ch
            self.getc()
        elif self.ch in TKS.PARANTH:
            self.sym = TKS.PARANTH[self.ch]
            self.value = self.ch
            self.getc()
        elif self.ch in TKS.BRACKETS:
            self.sym = TKS.BRACKETS[self.ch]
            self.value = self.ch
            self.getc()
        elif self.ch in TKS.SYMBOLS:
            self.sym = TKS.SYMBOLS[self.ch]
            self.value = self.ch
            self.getc()
        elif self.ch.isdigit():
            self.value = self.lex_num()
            self.sym = TKS.NUM
        elif self.ch.isalpha():
            word = ""
            while self.ch.isalpha() or self.ch.isdigit():
                word += self.ch
                self.getc()
            if word in TKS.WORDS:
                self.sym = TKS.WORDS[word]
                self.value = word
                break
            elif word in TKS.MATHFUNC:
                self.sym = TKS.MATHFUNC[word]
                self.value = word
                break
            elif word in TKS.METHODS:
                self.sym = TKS.METHODS[word]
                self.value = word
                break
            else:
                self.sym = TKS.VAR

```

```

        self.value = word
    else:
        self.error(ERR.LEX_ERRORS[0].format(self.ch))
class Node:
    """
    Building Tree o nodes by Parser object
    """
    def __init__(self):
        self.node_tree = {}
    def add_too_tree(self, key, value):
        self.node_tree[key] = value
class Parser:
    """
    Parsing token by token and building node tree
    """
    BEGIN = range(1)
    def __init__(self, lexer, node):
        self.lexer = lexer
        self.node = node
        self.parse()
    def error(self, msg):
        char_no = self.lexer.raw_text_file.tell()
        print('Parser error: ' + msg + "Символ номер " + str(char_no))
        f = open('pars_err.txt', 'w')
        PARS_ERROR = 'Parser error: ' + msg + "Символ номер " + str(char_no)
        f.write(PARS_ERROR)
        f.close()
        sys.exit(1)
    def eol_chek(self):
        """
        Chek End Of Line symbol ';'
        :return:
        """
        self.lexer.next_token()
        token = self.lexer.sym
        if token != TKS.SEMICOLON:
            self.error('Пропущен символ конца строки ";", в позиции ')
    def check_equ(self, varname, oper):
        self.lexer.next_token()
        if self.lexer.sym == TKS.EQUAL:
            pass
        else:
            self.error(ERR.PARS_ERRORS['Equal'].format(varname, oper))
    def check_varname(self):
        self.lexer.next_token()
        if self.lexer.sym == TKS.VAR:
            return self.lexer.value
        else:
            self.error('Ошибка в определении имени переменной. ')
    def check_num(self):
        self.lexer.next_token()
        if self.lexer.sym == TKS.NUM:
            return self.lexer.value
        else:
            self.error('Ошибка в определении значения переменной. ')
    def colon_check(self, operator):

```

```

"""
Chek COLON
:return:
"""

self.lexer.next_token()
token = self.lexer.sym
if token != TKS.COLON:
    self.error('Пропущен символ присваивания ":" оператора "{}".
'.format(operator))
def rpar_chek(self, key):
    value = ""
    while self.lexer.sym != TKS.RPAR:
        # self.parse_expression(key)
        self.lexer.next_token()
        value += self.lexer.value
        if self.lexer.sym == TKS.COMMA or self.lexer.sym == TKS.SEMICOLON:
            self.error('Пропущенна закрывающая скобка при определении "{}".
'.format(key))
    return value
def parse_expression(self, key):
    value = ""
    if self.lexer.ch in TKS.MATHOP: self.error(ERR.PARS_ERRORS['Expr'][2].format(key,
self.lexer.ch))
    while self.lexer.ch not in ',;':
        self.lexer.next_token()
        if self.lexer.sym == TKS.NUM:
            if self.lexer.ch in TKS.MATHOP or self.lexer.ch in ' ,;':
                value += self.lexer.value
            else:
                self.error(ERR.PARS_ERRORS['Expr'][7].format(self.lexer.ch))
        elif self.lexer.sym == TKS.VAR:
            if self.lexer.ch in TKS.MATHOP or self.lexer.ch in ' ,;':
                value += self.lexer.value
            else:
                self.error(ERR.PARS_ERRORS['Expr'][8].format(self.lexer.ch))
        elif self.lexer.sym == TKS.RPAR:
            value += self.lexer.value
        elif self.lexer.sym == TKS.LPAR:
            value += self.lexer.value + self.rpar_chek(key)
        elif self.lexer.value in TKS.MATHFUNC:
            if self.lexer.ch != '(':
                self.error(ERR.PARS_ERRORS['Expr'][3].format(key, self.lexer.value))
            else:
                value += self.lexer.value + self.rpar_chek(key)
        elif self.lexer.value in TKS.MATHOP:
            if self.lexer.ch in TKS.MATHOP:
                self.error(ERR.PARS_ERRORS['Expr']
[6].format(self.lexer.value,self.lexer.ch, key))
            else:
                value += self.lexer.value
        else:
            self.error(ERR.PARS_ERRORS['Expr'][5].format(self.lexer.value, key))
    if value[-1] in TKS.MATHOP:
        self.error(ERR.PARS_ERRORS['Expr'][4].format(key, value[-1]))
    return value
def check_diffname(self):

```

```

self.lexer.next_token()
if not self.lexer.value.startswith('d'):
    self.error(ERR.PARS_ERRORS['Expr'][1].format(self.lexer.value[0]))
else:
    return self.lexer.value
def parse_expressions(self):
    self.lexer.next_token()
    op = self.lexer.value
    if self.lexer.sym != TKS.EXPR:
        self.error(ERR.PARS_ERRORS['Expr'][0])
    self.colon_check(op)
    expr_dict = {}
    while self.lexer.ch != ';':
        k = self.check_diffname()
        self.check_equ(k, op)
        v = self.parse_expression(k)
        if v.count('(') > v.count(')'): self.error(
            'Ошибка в определении "{}". "(" больше чем ")" '.format(k))
        elif v.count('(') < v.count(')'): self.error(
            'Ошибка в определении "{}". "(" меньше чем ")" '.format(k))
        expr_dict[k] = v
        if self.lexer.ch == ',':
            self.lexer.next_token()
        else:
            break
    self.eol_chek()
    self.node.add_too_tree('Expr', expr_dict)
def parse_vars0(self):
    """
    Parsing Vars0 statement, looks like Vars0: y = 1, x = 1, z=0.5;
    or <НАЧ3НАЧ> = <ИНИЦПЕРЕМ> [", " {/ИНИЦПЕРЕМ/}] ";" on EBNF notation
    Full EBNF look in ebnf.txt
    :return: Finally make add_too_tree method
    """
    self.lexer.next_token()
    op = self.lexer.value
    if self.lexer.sym != TKS.VARS0:
        self.error(ERR.PARS_ERRORS['Vars0'][0])
    self.colon_check(op)
    vars_dict = {}
    while self.lexer.ch != ';':
        k = self.check_varname()
        self.check_equ(k, op)
        v = self.check_num()
        vars_dict[k] = v
        if self.lexer.ch == ',':
            self.lexer.next_token()
        else:
            break
    self.eol_chek()
    self.node.add_too_tree('Vars0', vars_dict)
def parse_coeff(self):
    """
    Parsing Coeff statement, looks like Coeff: asad = 2.0, b=56, c=8.098;;
    or <КОЭФ> = <ИНИЦПЕРЕМ> [", " {/ИНИЦПЕРЕМ/}] ";" on EBNF notation
    Full EBNF look in ebnf.txt

```

```

:return: Finally make add_too_tree method
"""
self.lexer.next_token()
op = self.lexer.value
if self.lexer.sym != TKS.COEFF:
    self.error(ERR.PARS_ERRORS['COEFF'][0])
self.colon_check(op)
coeff_dict = {}
while self.lexer.ch != ';':
    k = self.check_varname()
    self.check_equ(k, op)
    v = self.check_num()
    coeff_dict[k] = v
    if self.lexer.ch == ',':
        self.lexer.next_token()
    else:
        break
self.eol_chek()
self.node.add_too_tree('Coeff', coeff_dict)
def parse_step(self):
    """
    Parsing STEP statement, look like Step: 0.05;
    or <ШАГ> = "Step:" <ЧИСЛ> ";" on EBNF notation
    :return: Finally make add_too_tree method
    """
    self.lexer.next_token()
    op = self.lexer.value
    token = self.lexer.sym
    if token != TKS.STEP:
        self.error(ERR.PARS_ERRORS['STEP'][0])
    self.colon_check(op)
    self.lexer.next_token()
    step_value = self.lexer.value
    self.eol_chek()
    self.node.add_too_tree('Step', step_value)
def pars_range(self):
    """
    Parsing Range statment, look like [ NUM , NUM]
    or <ИНТЕРВАЛ> ::= "Range:" "[" <ЧИСЛ> "," <ЧИСЛ> "]" ";" in EBNF notation
    :return: Finally make add_too_tree method
    """
    self.lexer.next_token()
    op = self.lexer.value
    token = self.lexer.sym
    if token != TKS.RANGE:
        self.error(ERR.PARS_ERRORS['RANGE'][0])
    self.colon_check(op)
    self.lexer.next_token()
    if self.lexer.sym != TKS.LSB:
        self.error(ERR.PARS_ERRORS['RANGE'][1]
                    .format(self.lexer.value))
    else:
        range_value = self.lexer.value
        while self.lexer.sym != TKS.RSB:
            if self.lexer.sym == TKS.SEMICOLON:
                self.error(ERR.PARS_ERRORS['RANGE'][2].format(self.lexer.value))

```

```

        elif self.lexer.sym == TKS.NUM or self.lexer.value in '[,]':
            self.lexer.next_token()
        else:
            self.error(ERR.PARS_ERRORS['RANGE'][3].format(self.lexer.value))
            range_value += self.lexer.value
            for l in '[,]':
                if range_value.count(l) > 1: self.error(ERR.PARS_ERRORS['RANGE']
[4].format(l))
            self.node.add_too_tree('Range', range_value) # Add too tree
            self.eol_chek()
def pars_method(self):
    """
    Parsing Method statement, look like Method: MMethodName
    or <METOD> = "Method:" "Euler"|"RungeKutta2"|"RungeKutta4" ";" in EBNF form
    :return: Finally make add_too_tree method
    """
    self.lexer.next_token()
    op = self.lexer.value
    if self.lexer.sym != TKS.METHOD:
        self.error(ERR.PARS_ERRORS['METHOD'][0])
    self.colon_check(op)
    self.lexer.next_token()
    token = self.lexer.value
    if self.lexer.value not in TKS.METHODS:
        self.error(ERR.PARS_ERRORS['METHOD'][1].format(self.lexer.value))
    method_value = self.lexer.value
    self.eol_chek()
    self.node.add_too_tree('Method', method_value)
def statement(self):
    if self.lexer.sym != TKS.BEGIN:
        self.error('Программа должна начинаться с оператора Begin:, а пришло {}'.
            .format(self.lexer.value))
    self.eol_chek()
    self.parse_expressions()
    self.parse_vars0()
    self.parse_coeff()
    self.parse_step()
    self.pars_range()
    self.pars_method()
    self.lexer.next_token()
    if self.lexer.sym != TKS.END:
        self.error('Программа должна заканчиваться оператором End, а пришло {}'.
            .format(self.lexer.value))
    self.eol_chek()
def parse(self):
    self.lexer.next_token()
    self.statement()
    self.lexer.next_token()
    if (self.lexer.sym != TKS.EOF):
        self.error('Неправильный синтаксис выражения')
    else:
        print('OK')

```



```

==> post_lex_handler.py <==
import sys
import TOKENS as TKS
import ERRORS as ER
class PostParserHandler:
    def __init__(self, nodetree):
        self.PLH_ERROR=""
        self.nt = nodetree
        self.lexems = [k for l in [TKS.WORDS.keys(), TKS.MATHFUNC.keys(),
TKS.METHODS.keys()] for k in l]
        self.chek_vars_and_diffs()
        self.replace_coeff()
        self.replace_pow()
    def error(self, msg):
        print('Ошибка постобработчика: ' + msg)
        f=open('plh_err.txt','w')
        PLH_ERROR = 'Ошибка постобработчика: ' + str(msg)
        f.write(PLH_ERROR)
        f.close()
        sys.exit(1)
    def chek_vars_and_diffs(self):
        if len(set(self.nt['Expr'].keys())) != len(set(self.nt['Vars0'].keys())):
            self.error(
                ER.POST_PARS_ERRORS[0].format(len(set(self.nt['Expr'].keys())),
len(set(self.nt['Vars0'].keys()))))
            for dif in [k[1] for k in set(self.nt['Expr'].keys())]:
                if dif not in set(self.nt['Vars0'].keys()):
                    self.error(ER.POST_PARS_ERRORS[1].format(dif))
            for dif in [k[1] for k in set(self.nt['Expr'].keys())]:
                if dif in set(self.nt['Coeff'].keys()):
                    self.error(ER.POST_PARS_ERRORS[2].format(dif, dif))
    def replace_coeff(self):
        """
        Замена коэффициентов их значениями, если всё успешно .pop('Coeff')
        :return: None
        """
        for k, v in self.nt['Expr'].items():
            for coef in self.nt['Coeff'].items():
                v = v.replace(coef[0], coef[1])
            chars_only = [char for char in v.replace('sin', "").replace('cos',
'').replace('pi', "").replace('tg', '')
                if char.isalpha()]
            for char in chars_only:
                if char not in self.nt['Vars0']:
                    self.error(ER.POST_PARS_ERRORS[3].format(k, char))
            self.nt['Expr'][k] = v
        self.nt.pop('Coeff')
    def replace_sin(self):
        result = self.nt
        for v in result['Expr'].items():
            if 'sin' in v[1]:
                newstr = v[1].replace('sin', 'math.sin')
                result['Expr'][v[0]] = newstr
        self.nt = result
    def replace_cos(self):
        result = self.nt

```

```

    for v in result['Expr'].items():
        if 'cos' in v[1]:
            newstr = v[1].replace('cos', 'math.cos')
            result['Expr'][v[0]] = newstr
    self.nt = result
def replace_tg(self):
    result = self.nt
    for v in result['Expr'].items():
        if 'tg' in v[1]:
            newstr = v[1].replace('tg', 'tan')
            result['Expr'][v[0]] = newstr
    self.nt = result
def replace_pi(self):
    result = self.nt
    for v in result['Expr'].items():
        if 'pi' in v[1]:
            newstr = v[1].replace('pi', 'math.pi')
            result['Expr'][v[0]] = newstr
    self.nt = result
def replace_pow(self):
    result = self.nt
    for v in result['Expr'].items():
        if '^' in v[1]:
            newstr = v[1].replace('^', '**')
            result['Expr'][v[0]] = newstr
    self.nt = result

```

```

==> rgkt4.py <==
import matplotlib.pyplot as plt
import numpy as np
from math import sin as sin
from math import cos as cos
from math import tan as tan
from math import pi as pi
class RgKt4:
def getChr(i):
    return chr(i + 97)

def genRK(n):
    print("# fourth order Runge-Kutta method in " + str(n) + " dimensions")
    u = ""
    v = ""
    f = ""
    for i in range(n):
        c = getChr(i)
        if i != 0:
            u += ", "
            v += ", "
            f += ", "
        u += c
        v += c + "k"
        f += "f" + c
    print("def rK" + str(n) + "(" + u + ", " + f + ", hs" + "):")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "1 = f" + c + "(" + u + ")*hs")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "k = " + c + " + " + c + "1*0.5")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "2 = f" + c + "(" + v + ")*hs")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "k = " + c + " + " + c + "2*0.5")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "3 = f" + c + "(" + v + ")*hs")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "k = " + c + " + " + c + "3")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + "4 = f" + c + "(" + v + ")*hs")
    for i in range(n):
        c = getChr(i)
        print("\t" + c + " = " + c + " + (" + c + "1 + 2*(" + c + "2 + " + c + "3) + "
+ c + "4)/6")
    print("\treturn " + u)

```

```

==> euler.py <==
import matplotlib.pyplot as plt
import numpy as np
from math import sin as sin
from math import cos as cos
from math import tan as tan
from math import pi as pi
class EulerMethod:
    def __init__(self, **kwargs):
        rang0 = eval(kwargs['Range'])[0]
        rang1 = eval(kwargs['Range'])[1]
        self.rang0 = rang0
        self.rang1 = float(rang1)
        self.step = float(kwargs['Step'])
        self.express = kwargs['Expr']
        self.time_arr = np.linspace(self.rang0,
                                    self.rang1,
                                    int(self.rang1 / self.step) + 1)
        self.vars0 = {k: eval(v) for k, v in kwargs['Vars0'].items()}
        self.res = {}
        self.gen_zeros()
        self.gen_res()
    def gen_zeros(self):
        for k, v in self.vars0.items():
            self.res[k] = np.zeros(len(self.time_arr))
            self.res[k][0] = v
    def gen_res(self):
        for i in range(1, len(self.time_arr)):
            for k, v in self.res.items():
                dic = {k: v[i - 1] for k, v in self.res.items()}
                dic['sin'] = sin
                dic['cos'] = cos
                dic['tg'] = tan
                dic['pi'] = pi
                self.res[k][i] = round(self.res[k][i - 1] + \
                                       eval(self.express['d' + k],
                                             dic) \
                                       * self.step, 3)
    def gen_plot(self):
        color = ['red', 'green', 'blue', 'yellow']
        i = 0
        plt.figure()
        for k in self.res:
            plt.plot(self.time_arr, self.res[k], color=color[i])
            i += 1
        plt.xlabel('t')
        plt.show()
    def printval(self):
        self.res['t'] = self.time_arr
        return self.res
    def gen_arr(self):
        sting_arr = ""
        sting_arr += "t"
        for char in self.time_arr:
            sting_arr += "|{: ^10}".format(round(float(char), 2))
        sting_arr += '\n'

```

```
for k, v in self.res.items():
    sting_arr += k
    for l in v:
        sting_arr += "|{: ^10}".format(round(float(l), 2))
    sting_arr += '\n'
print(sting_arr)
return sting_arr
```

```

==> run.py <==
import os
import sys
from tkinter import *
import matplotlib.pyplot as plt
from lexex import Node as Node
from lexex import Lexer as Lexer
from lexex import Parser as Parser
from lexex import PreLexer as PreLexer
from euler import EulerMethod as EulerMethod
from post_lex_handler import PostParserHandler as PostParserHandler
class MyUI(Tk):
    def __init__(self):
        super().__init__()
        self.res_arr = {}
        self.time_arr = []
        self.initUI()
        self.Pack()
    def initUI(self):
        self.f = Frame(self)
        self.title('Анализатор')
        self.text_area = Text(self.f, height=15, width=140, font='Monospace 12',
undo=True)
        self.xscroll = Scrollbar(self.f, orient=HORIZONTAL)
        self.result = Text(self.f, width=140, font='Monospace 12', height=3,
background='#ccc', wrap=NONE,
xscrollcommand=self.xscroll.set)
        st = ''
        self.result.insert(INSERT, st)
        self.result.configure(xscrollcommand=self.xscroll.set)
        self.xscroll.config(command=self.result.xview)
        self.result.configure(state=DISABLED)
        self.btn_doit = Button(self.f, text='DO IT!', command=self.do_it)
        self.show_plot = Button(self.f, text='Show Plot', command=self.gen_plot,
state=DISABLED)
    def Pack(self):
        self.f.pack()
        self.text_area.pack()
        self.btn_doit.pack(fill=X)
        self.show_plot.pack()
        self.result.pack(fill=X)
        self.xscroll.pack(fill=X)
    def gen_plot(self):
        color = ['red', 'green', 'blue', 'yellow']
        i = 0
        plt.figure()
        for k in self.res_arr:
            plt.plot(self.time_arr, self.res_arr[k], color=color[i], label='d'+k)
            i += 1
        plt.xlabel('t')
        plt.grid(True)
        plt.legend(loc='lower right')
        plt.show()
        print(self.time_arr)
    def magic(self):
        file = 'input.txt'
        prelexer=PreLexer(file)

```

```

lexer = Lexer(file)
node = Node()
p = Parser(lexer, node)
FINALNODETREE = p.node.node_tree
p = PostParserHandler(FINALNODETREE)
result = p.nt
METHOD = result.pop('Method')
d = EulerMethod(**result)
OUTPUT = d.gen_arr()
self.res_arr = d.printval()
self.time_arr = self.res_arr.pop('t')
print(self.res_arr)
return OUTPUT
def do_it(self):
    file = open('input.txt', 'w')
    file.write(self.text_area.get('1.0', END))
    file.close()
    try:
        try:
            self.text_area.tag_delete('ER')
        except:
            pass
        out = self.magic()
        self.result.configure(state=NORMAL)
        self.result.delete('1.0', END)
        self.result.insert(END, out)
        self.result.configure(state=DISABLED, background='#a8ffcd')
        self.show_plot.configure(state=NORMAL)
    except:
        self.result.configure(state=NORMAL)
        self.result.delete('1.0', END)
        if os.path.isfile('pre_lex_err.txt'):
            err = open('pre_lex_err.txt').readline()
            l_n = int(err.split()[-1])
            self.text_area.tag_configure('ER', background='#ffa8a8')
            self.text_area.tag_add('ER', '{}.0'.format(l_n), '{}.0 lineend'.format(l_n))
            os.remove('pre_lex_err.txt')
        elif os.path.isfile('plh_err.txt'):
            err = open('plh_err.txt').readline()
            os.remove('plh_err.txt')
            self.text_area.tag_configure('ER', background='#ffa8a8')
            self.text_area.tag_add('ER', '2.0', '2.0 lineend')
        elif os.path.isfile('pars_err.txt'):
            err = open('pars_err.txt').readline()
            char_no= int(err.split()[-1])
            self.text_area.tag_configure('ER', background='#ff0000', foreground="#fff",
underline=True)
            self.text_area.tag_add('ER', "1.0+{}c".format(char_no-2))
            os.remove('pars_err.txt')
        elif os.path.isfile('lex_err.txt'):
            err = open('lex_err.txt').readline()
            char_no= int(err.split()[-1])
            self.text_area.tag_configure('ER', background='#ff0000', foreground="#fff",
underline=True)
            self.text_area.tag_add('ER', "1.0+{}c".format(char_no-2))
            os.remove('lex_err.txt')

```

```
        self.result.insert(END, err)
        self.result.configure(state=DISABLED, background='#ffa8a8')
if __name__ == '__main__':
    ui = MyUI()
    ui.mainloop()
```



```

==> ERRORS.py <==
PRE_LEX_ERRORS = ('Отсутствует символ конца строки ";" в строке номер {}',)
LEX_ERRORS = ('Непредвиденный символ "{}". Символ номер ',
              'Непредвиденный символ "{}" при определении вещественного числа. Символ номер ',
              'Очень много точек в вещественном числе "{}". Символ номер ',
              'При обработки вещественного числа, пришел символ новой строки "\n". ',
              'Скорее всего пропущен символ конца строки ";" Символ номер ',
              )
PARS_ERRORS = {"RANGE": ['Пропущен оператор определения интервала "Range". ',
                        'Ожидался открывающий символ "[" при определении "Range", а пришел "{}". ',
                        'Ожидался закрывающий символ "]" при определении "Range", а пришел "{}'. ',
                        'Непредвиденный символ "{}" при определении "Range". ',
                        'Лишний символ "{}" при определении "Range". ',
                        ],
               'METHOD': ['Пропущен оператор определения шага "Method". ',
                        'Неизвестный метод интегрирования {}'. ',
                        ],
               "STEP": ['Пропущен оператор определения шага "Step". '],
               "COEFF": ['Пропущен оператор определения коэффициентов "Coeff". '],
               "Vars0": ['Пропущен оператор определения начальных условий "Vars0". '],
               "Expr": ['Пропущен оператор определения дифференциальных уравнений "Expr". ',
                       'Определение дифференциального уравнения должно начинаться с символа "d", а пришел "{}'. ',
                       'Выражение "{}" не может начинаться со знака математической операции "{}'. ',
                       'Ошибка в определении "{}". После математической функций "{}" должна идти скобка. ',
                       'Ошибка определения "{}". Выражение не может заканчиваться знаком математической операции "{}'. ',
                       'Непредвиденный символ "{}" в определении "{}'. ',
                       'Два знака математических операций "{}{}" подряд в определении "{}'. ',
                       'После числа может идти матоператор или ', ';)', 'а пришел '{}'. ',
                       'После переменной может идти матоператор или ', ';)', 'а пришел '{}'. ', ],
               "Equal": ['Пропущен символ равенства "=" после переменной "{}" в операции "{}'. ',
                       ],
               }
POST_PARS_ERRORS = ('Определено "{}" дифференциальных уравнений. Начальные условия предоставлены для "{}'. ',
                   'Для "d{}" не определено начальное условие',
                   'Имя уравнения "d{}" совпадает с одним из коэффициентов "{}"',
                   'Ошибка в определении '{}'. Неизвестная переменная '{}. ',)

```