

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«МИРЭА—Российский технологический университет»**

**РТУ МИРЭА**

**Институт комплексной безопасности и специального приборостроения**

(наименование института, филиала)

Кафедра КБ-3 «Управление и моделирование систем»

(наименование кафедры)

**Курсовая работа**

по дисциплине \_\_\_\_\_\_\_\_\_\_\_\_Разработка проблемно-ориентированных\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_транслирующих средств\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(наименование дисциплины)

**Тема курсовой работы** \_\_\_\_\_\_\_\_\_\_\_Разработка программного обеспечения, реализующего транслирующее средство с проблемно-ориентированного языка

**Студент группы** БСБО-01-18 Горшков Максим Витальевич  **\_\_\_\_\_\_\_\_\_\_\_\_**

(учебная группа, фамилия, имя, отчество студента) (подпись студента)

**Руководитель курсовой работы\_\_\_\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_**

(должность, звание, ученая степень) (подпись руководителя)

**Рецензент** (при наличии)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(должность, звание, ученая степень) (подпись рецензента)

Работа представлена к защите «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

Допущен к защите «\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_г.

Оглавление

[Введение 1](#_Toc90339507)

[Форма Бэкуса-Наура для языка 2](#_Toc90339508)

[Блок-схемы языка 3](#_Toc90339509)

[Демонстрация работы программы 9](#_Toc90339510)

[Листинг кода программы 15](#_Toc90339511)

## Введение

**Цель работы:**

Разработка программного обеспечения, реализующего транслирующее средство с проблемно-ориентированного языка; закрепление навыков разработки сложных программных систем, включающих в себя элементы системного программирования.

**Постановка задачи (7 вариант):**

Разработать язык программирования согласно предложенной форме Бэкуса-Наура. Программное обеспечение, обрабатывающее этот язык, должно иметь синтаксический анализатор, проверяющий входной текст на наличие ошибок.

**Текст варианта:**

Язык = "Начало" Объявление...Объявление Опер";"...Опер "Конец"

Объявление = "Переменные" Перем...Перем |

"Метки" Цел...Цел "Конец меток”

Опер = Метка...Метка ":" Перем "=" Прав.часть

В правой части – переменные и целые числа, соединены аддитивными, мультипликативными и логическими операциями; есть круглые скобки любой степени вложенности.

Переменная – буква, затем до трёх цифр.

Русский алфавит, восьмеричная арифметика.

## Форма Бэкуса-Наура для языка

Язык = "Начало" Объявление...Объявление Опер";"...Опер "Конец"

Объявление = "Переменные" Перем...Перем |

"Метки" Цел...Цел "Конец меток"

Опер = Метка...Метка ":" Перем "=" Прав.часть

Прав.часть = </-/> блок1 ["+" | "-"]... блок1

блок1 = блок2 ["\*" | "/"]... блок2

блок2 = блок3 ["|" | "&"]... блок3

блок3 = </"!"/> блок4

блок4 = Цел | перем | "(" Прав.часть ")"

Перем = Бук </Циф/> </Циф/> </Циф/>

Метка = Цел

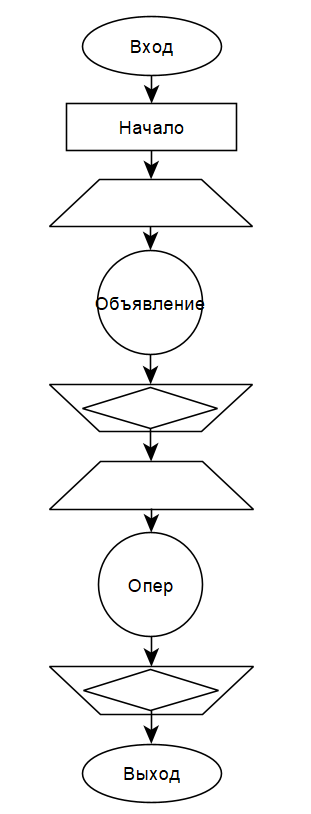
Цел = Циф ... Циф

Циф = "0"|"1"|...|"7"

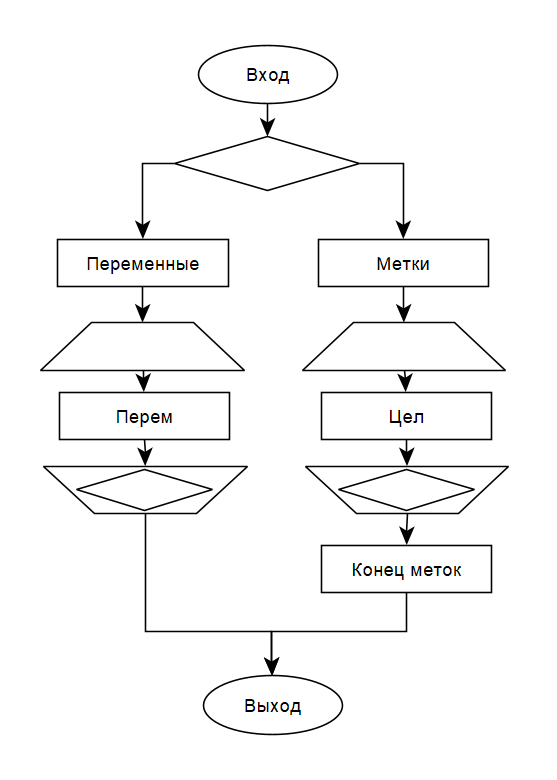
Бук = "А"|"Б"|...|"Я"|"а"|"б"|...|"я"

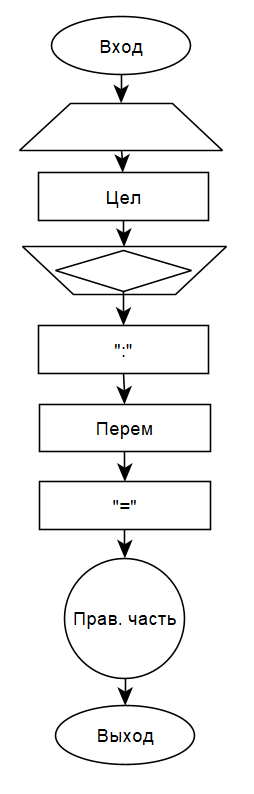
## Блок-схемы языка

**Язык**

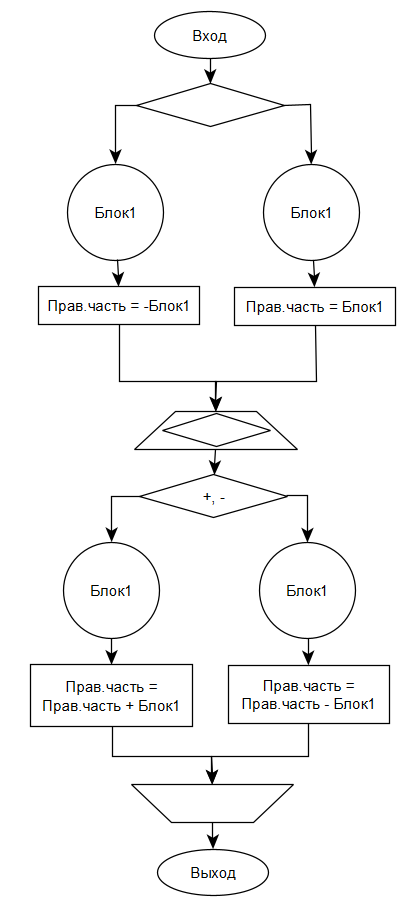


**Объявление**

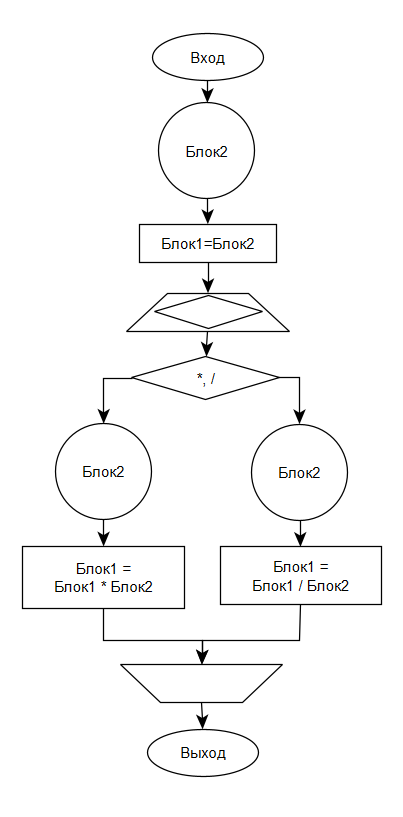
**Опер**



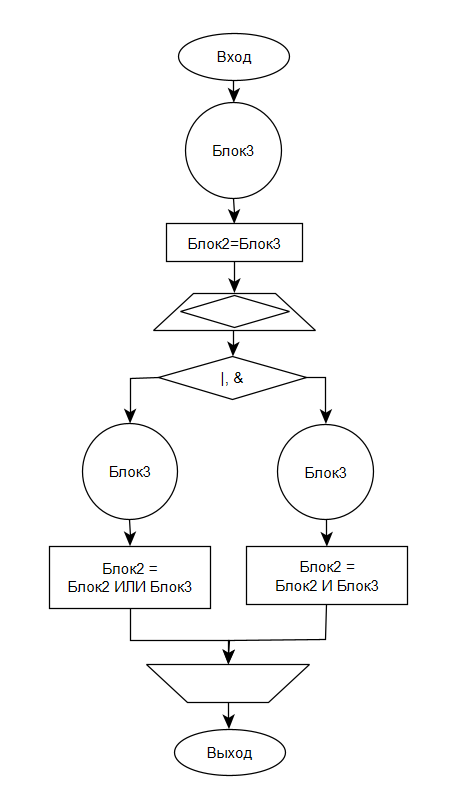
**Правая часть**



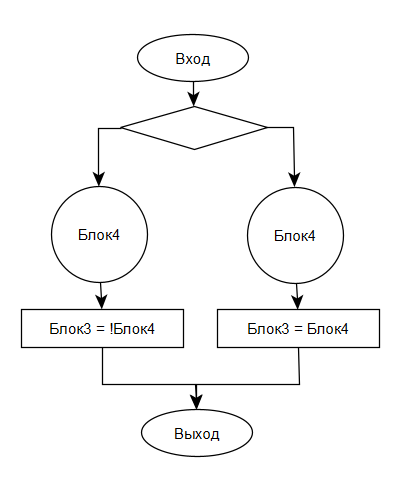
**Блок 1**



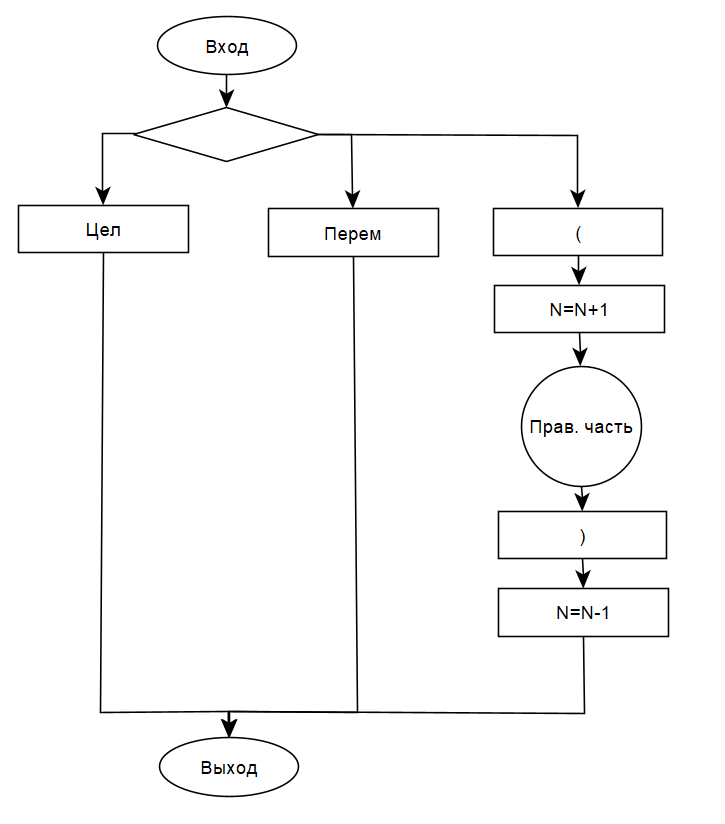
**Блок 2**



**Блок 3**

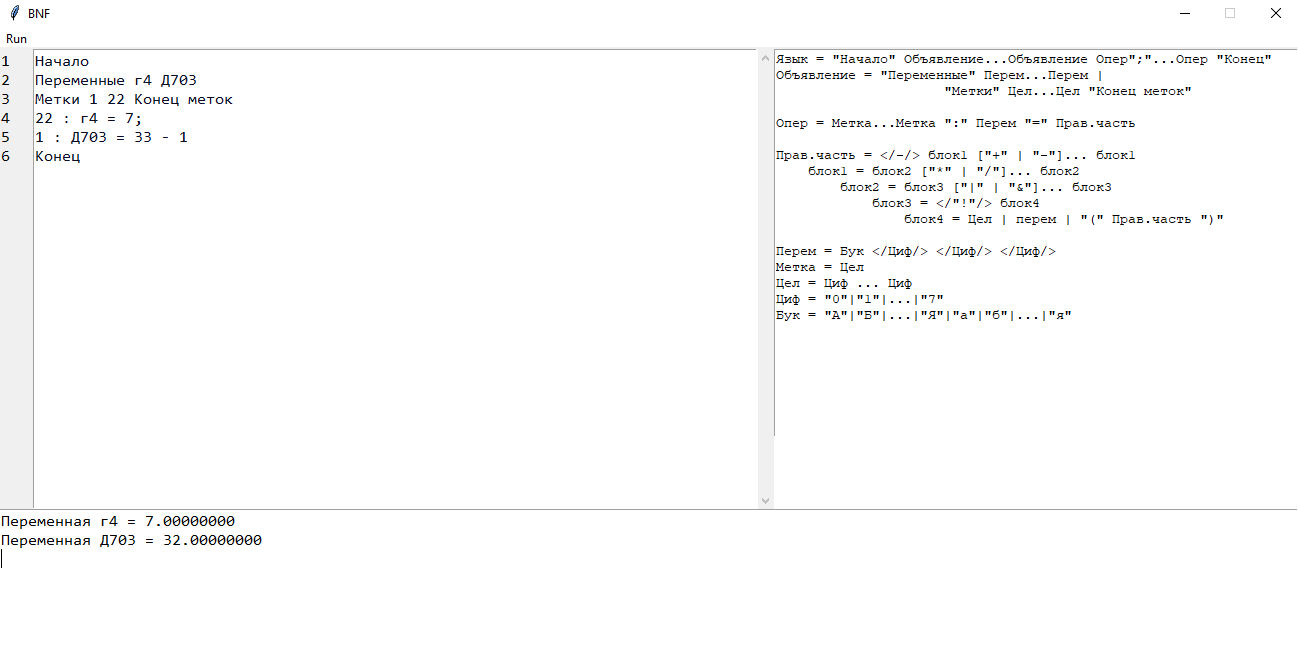


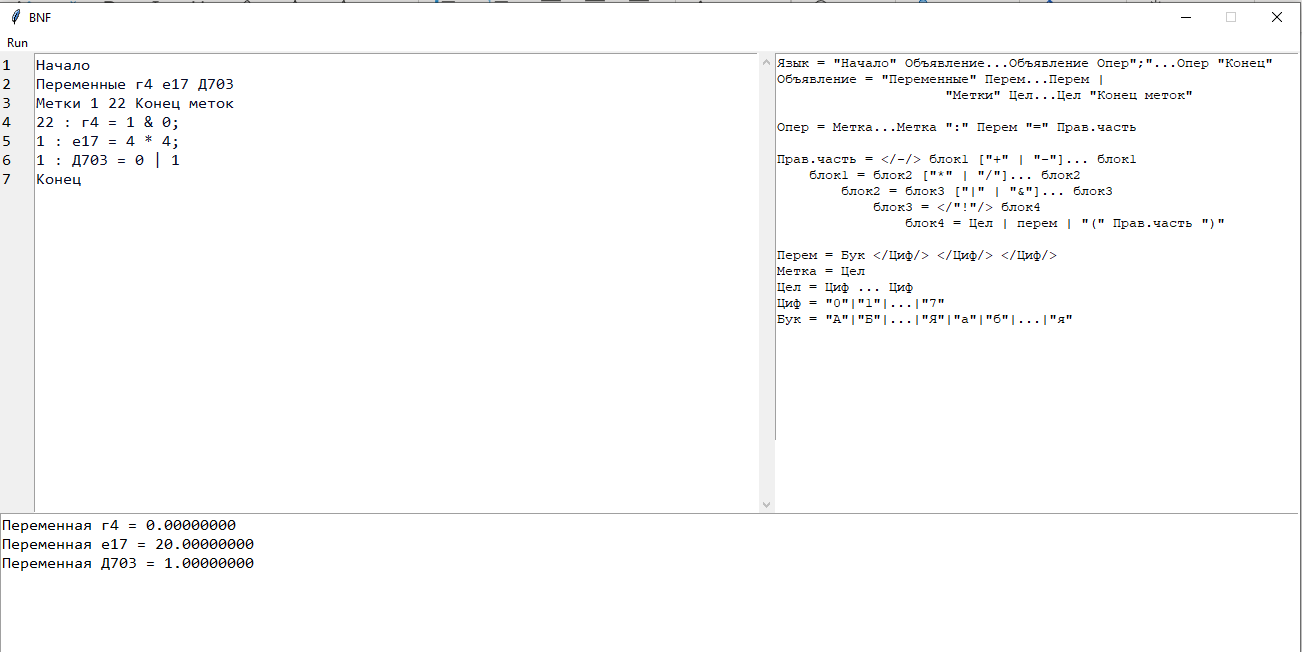
**Блок 4**

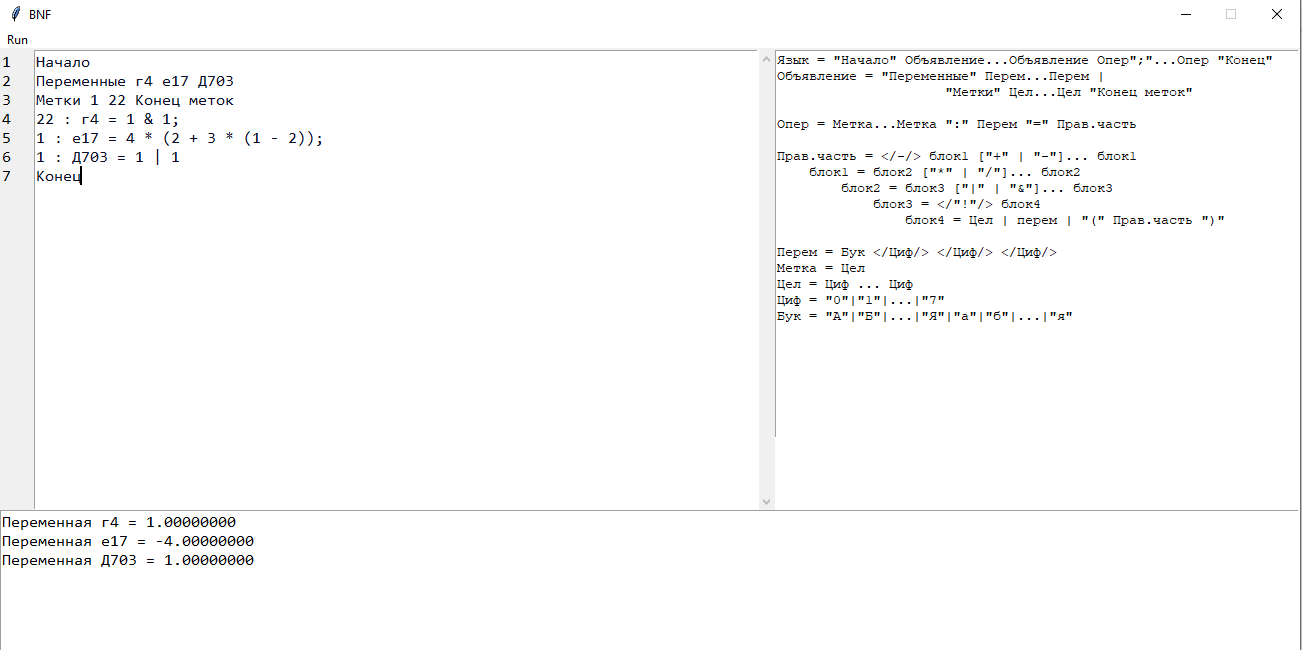


## Демонстрация работы программы

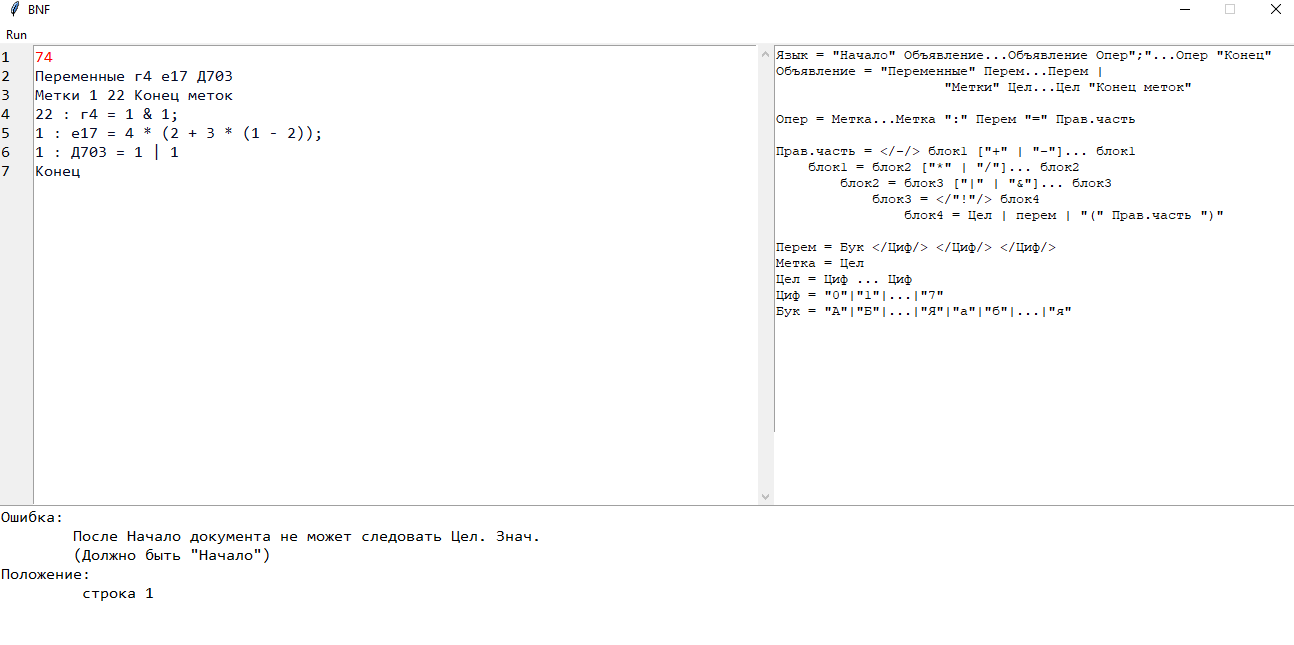
«Правильные» примеры на заданном языке

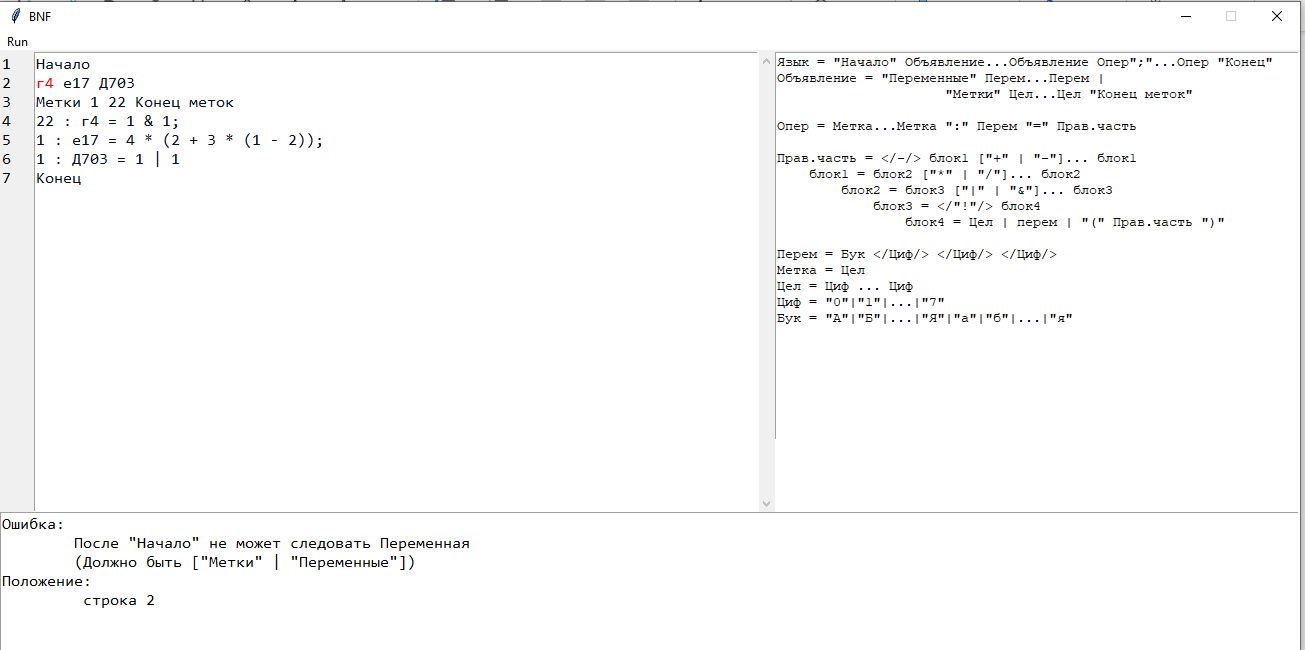


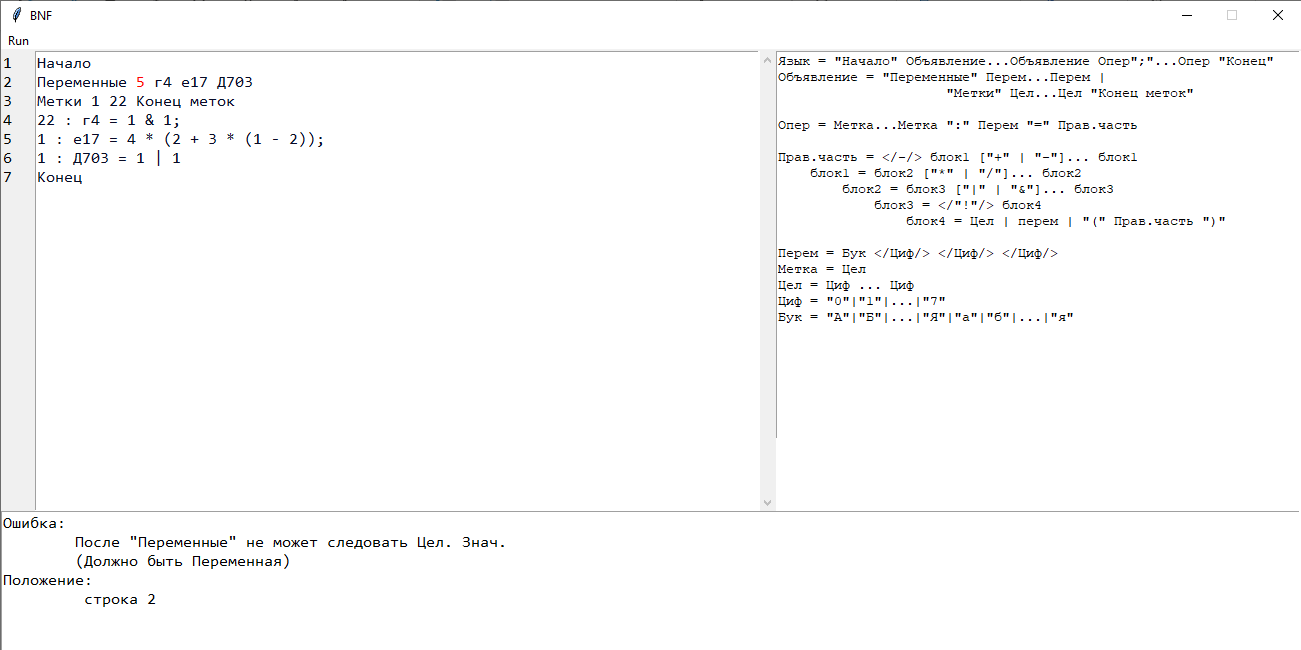


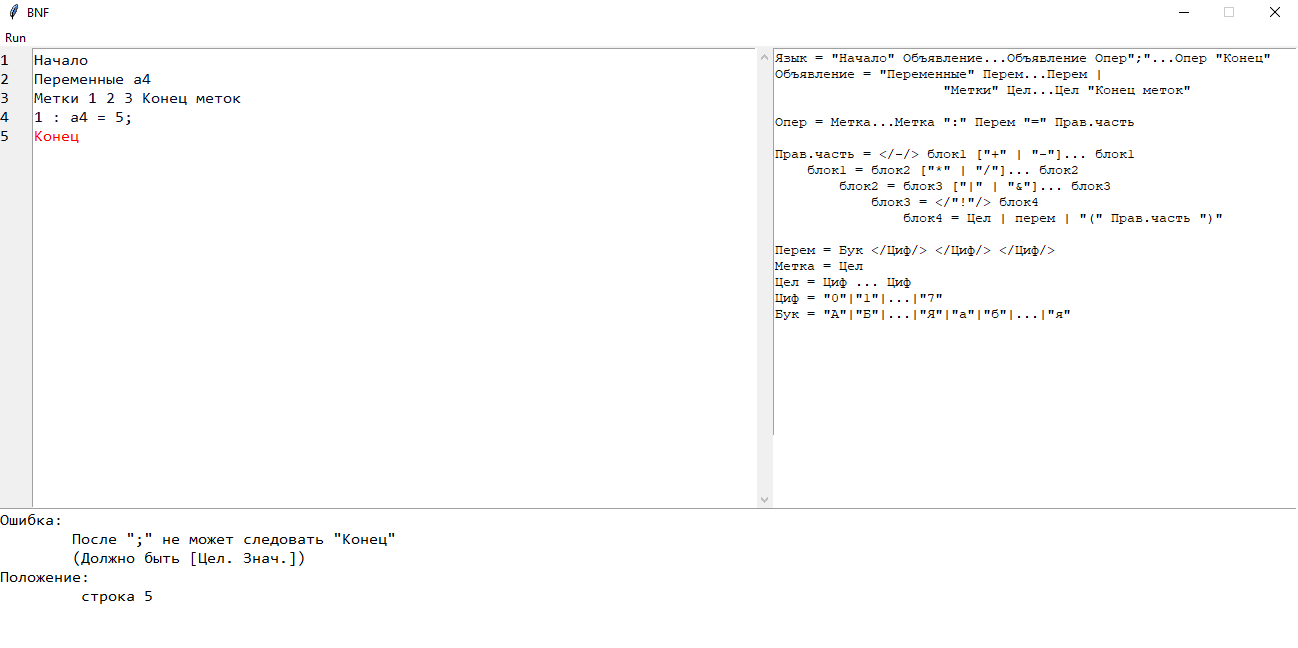
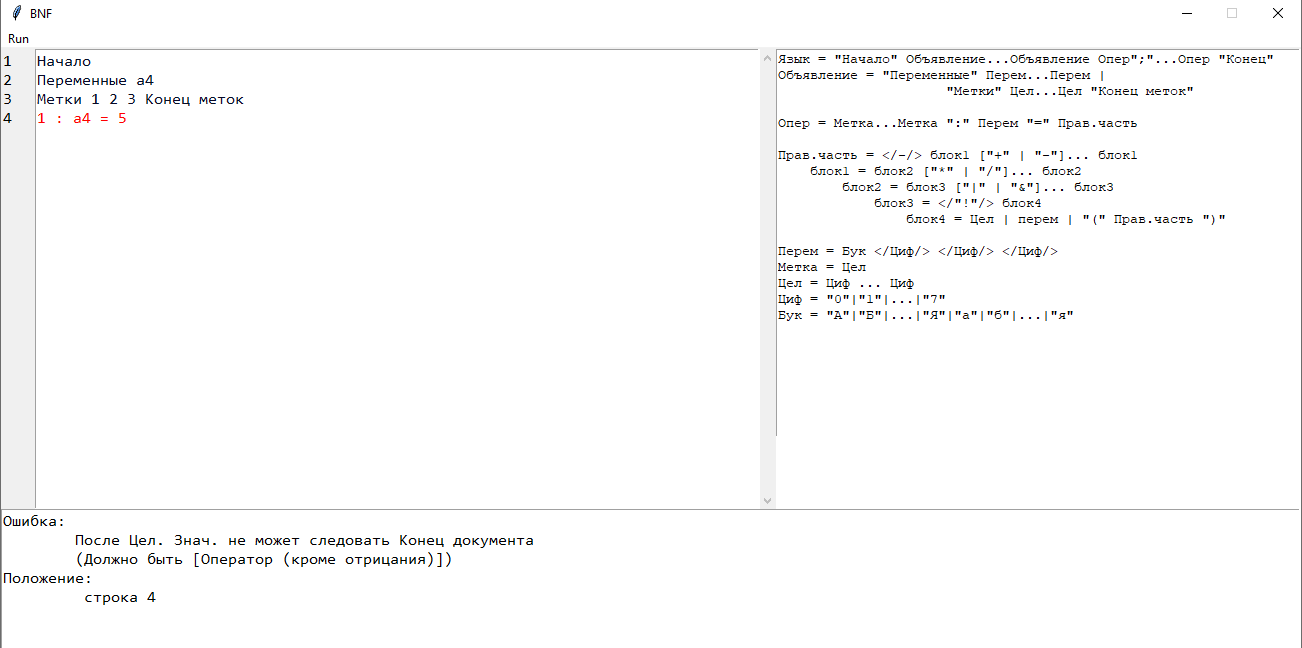
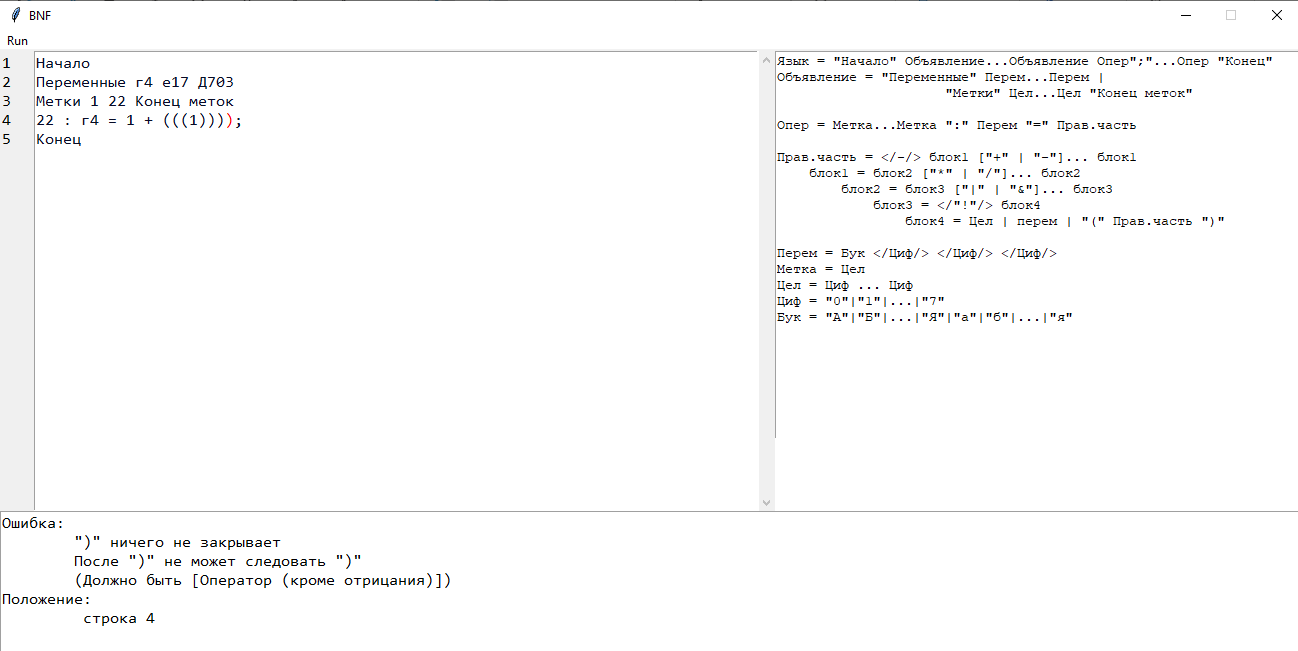
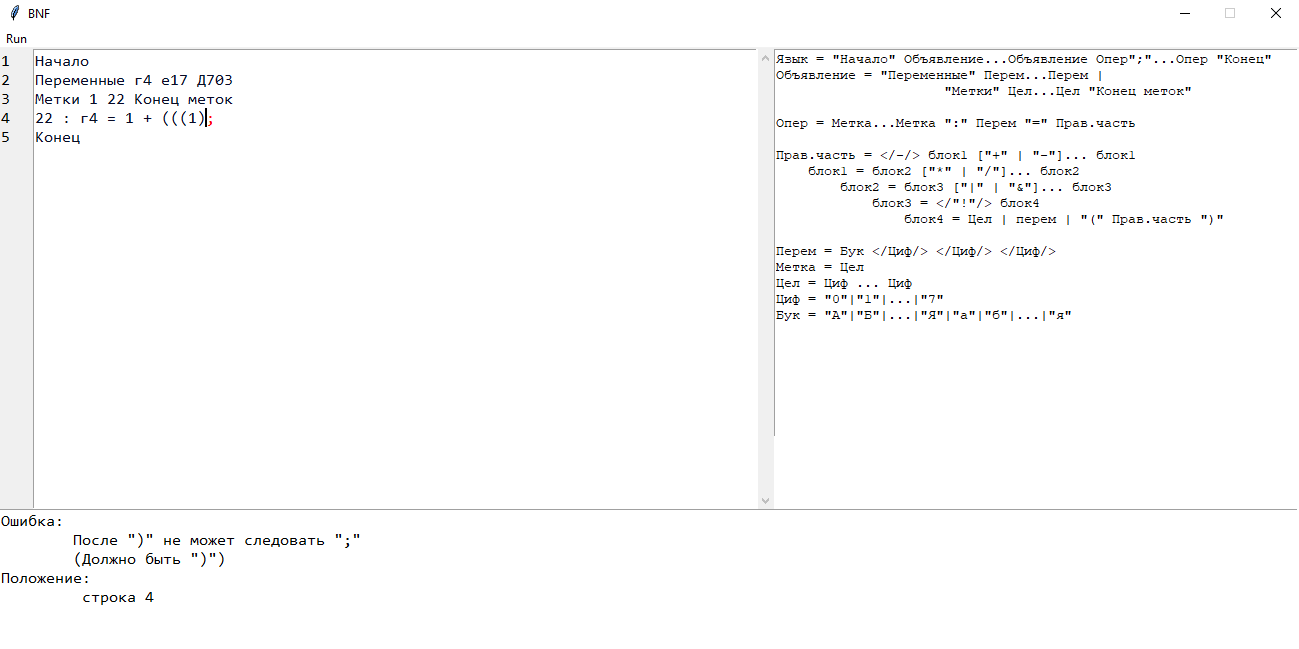
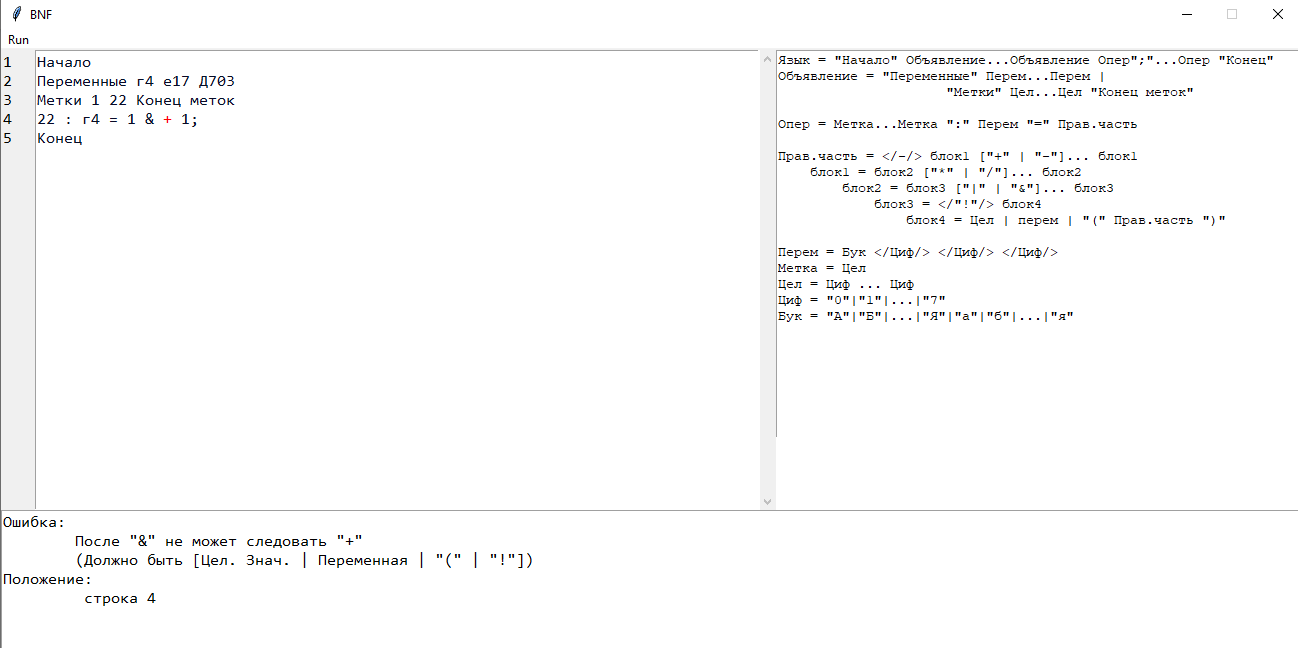
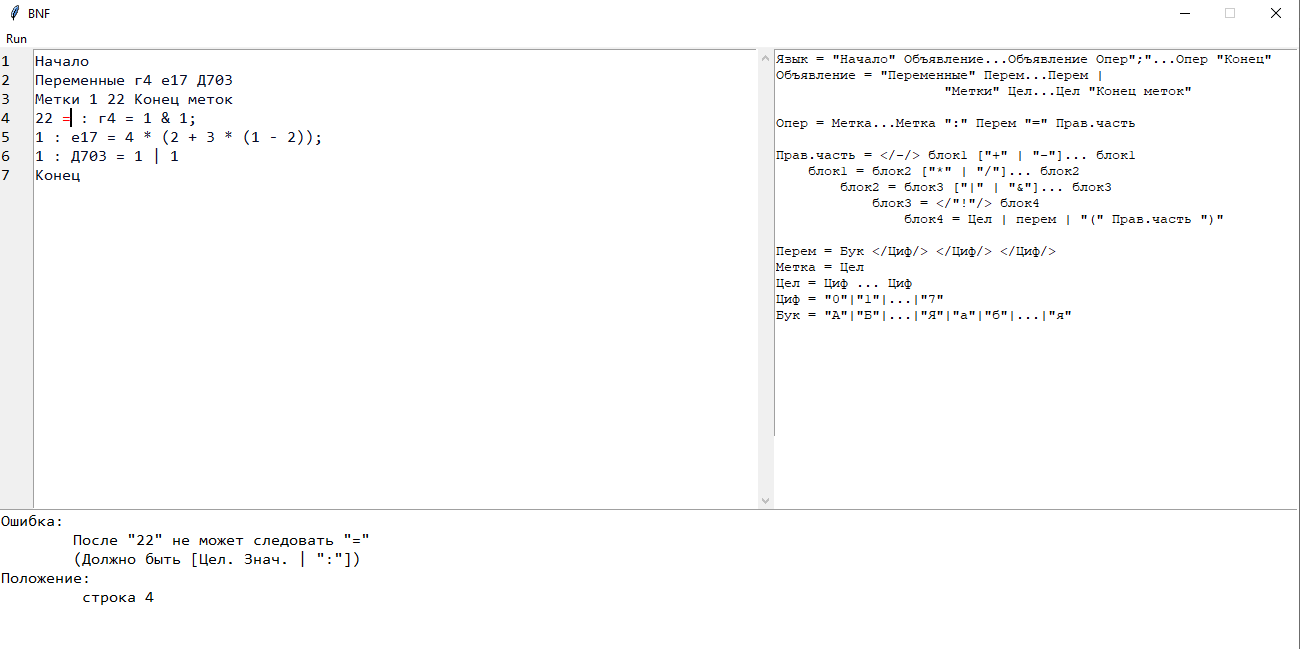
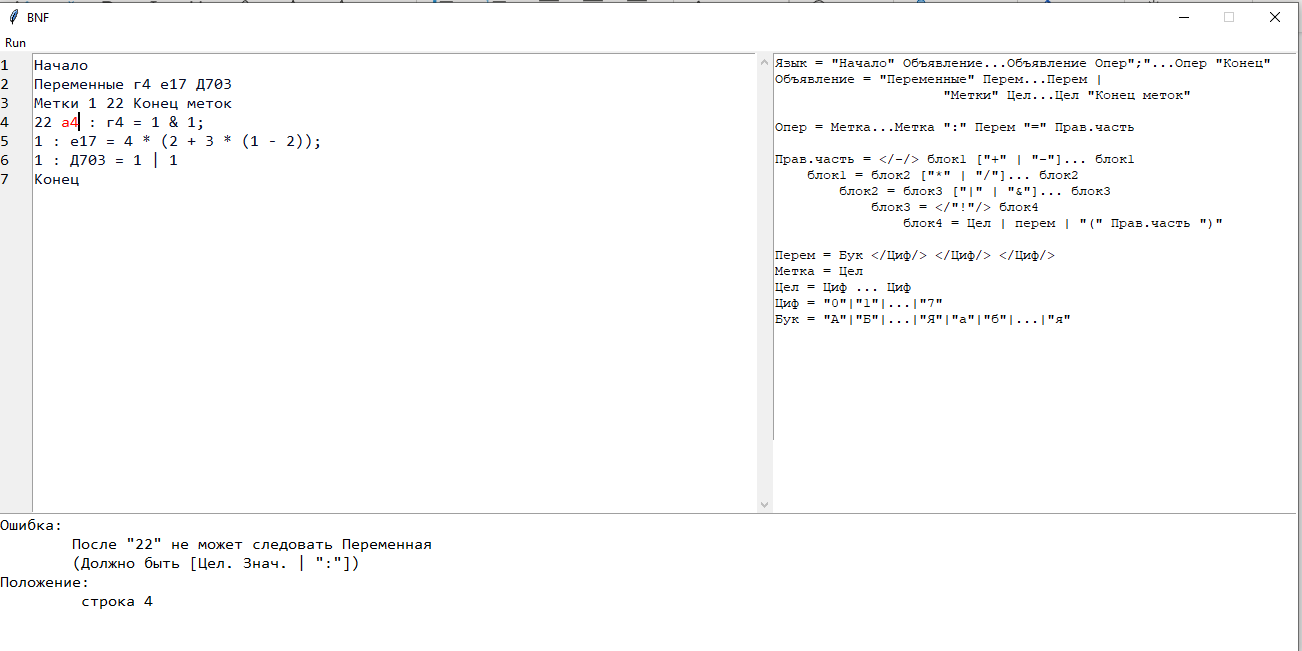
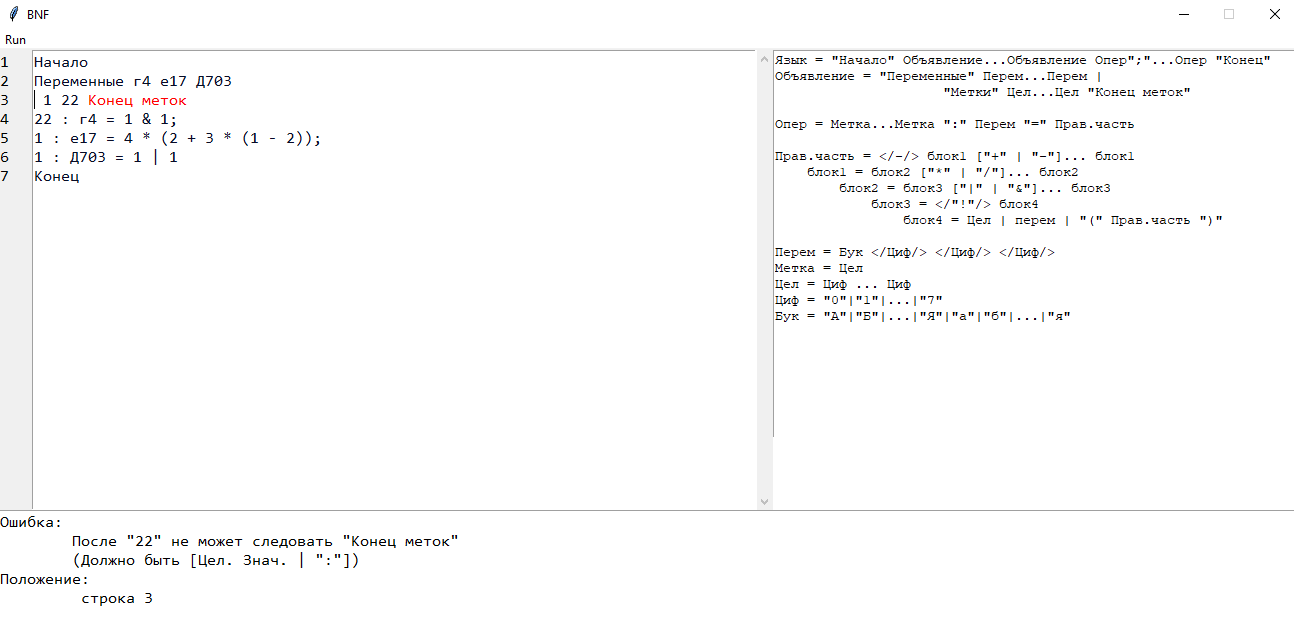


«Ошибочные» примеры на заданном языке









## Листинг кода программы

**Code\_input.py**

import tkinter as tk

class TextLineNumbers(tk.Canvas):

def \_\_init\_\_(self, font, \*args, \*\*kwargs):

tk.Canvas.\_\_init\_\_(self, \*args, \*\*kwargs)

self.textwidget = None

self.font = font

def attach(self, text\_widget):

self.textwidget = text\_widget

def redraw(self, \*args):

'''redraw line numbers'''

self.delete("all")

i = self.textwidget.index("@0,0")

while True :

dline= self.textwidget.dlineinfo(i)

if dline is None: break

y = dline[1]

linenum = str(i).split(".")[0]

self.create\_text(2, y, anchor="nw", text=linenum, font=self.font)

i = self.textwidget.index("%s+1line" % i)

class CustomText(tk.Text):

def \_\_init\_\_(self, \*args, \*\*kwargs):

tk.Text.\_\_init\_\_(self, \*args, \*\*kwargs)

self.\_orig = self.\_w + "\_orig"

self.tk.call("rename", self.\_w, self.\_orig)

self.tk.createcommand(self.\_w, self.\_proxy)

def \_proxy(self, command, \*args):

if command == 'get' and (args[0] == 'sel.first' and args[1] == 'sel.last') and not self.tag\_ranges('sel'): return

if command == 'delete' and (args[0] == 'sel.first' and args[1] == 'sel.last') and not self.tag\_ranges('sel'): return

cmd = (self.\_orig, command) + args

result = self.tk.call(cmd)

if command in ('insert', 'delete', 'replace'):

self.event\_generate("<<Change>>", when="tail")

return result

**Code\_parser.py**

from re import A

from typing import Iterable, List

from tokenizer import \*

from nodes import \*

from errors import CustomSyntaxError, SyntaxErrorExpectedVSGot, SyntaxErrorMaxBracketsDepth

class Parser():

def \_\_init\_\_(self, tokens: List, max\_brackets\_depth=50) -> None:

assert max\_brackets\_depth >= -1

self.tokens = tokens

self.max\_brackets\_depth = max\_brackets\_depth

self.reset()

def reset(self):

self.\_pos = 0

self.\_current = self.tokens[self.\_pos]

def next(self, n=1):

assert n >= 1, f"{n} < 1"

self.\_pos += n

if self.\_pos < len(self.tokens):

self.\_current = self.tokens[self.\_pos]

if self.\_current.type == "skip":

return self.next()

else:

self.\_current = None

return self.\_current

def prev(self, n=1):

assert n >= 1, f"{n} < 1"

self.\_pos -= n

if self.\_pos >= 0:

self.\_current = self.tokens[self.\_pos]

if self.\_current.type == "skip" and not self.\_current.token == T\_BOF:

return self.prev()

else:

self.\_current = None

return self.\_current

def ast(self):

self.reset()

try:

tree = self.program()

except (SyntaxErrorExpectedVSGot, SyntaxErrorMaxBracketsDepth) as e:

return None, e

else:

return tree, None

def named\_set(self, al\_one=False) -> NodeNamedSet:

if self.\_current.token == T\_VARS:

self.next()

set\_element = self.var\_declaration()

next\_named\_set = self.named\_set(True)

return NodeNamedSet(set\_element, next\_named\_set)

elif self.\_current.token == T\_MARKS:

self.next()

set\_element = self.mark\_declaration()

next\_named\_set = self.named\_set(True)

return NodeNamedSet(set\_element, next\_named\_set)

elif al\_one:

return None

raise SyntaxErrorExpectedVSGot([T\_MARKS, T\_VARS], self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

def term(self):

expected\_values = (T\_INT, T\_ID)

expected\_unary\_operators = (T\_NOT,)

if self.\_current.token in expected\_values:

term = self.\_current

self.next()

return term

elif self.\_current.token in expected\_unary\_operators:

operator = self.\_current

self.next()

term = self.term()

return NodeUnaryOperator(operator, term)

elif self.\_current.token == T\_LBR:

self.next()

term = self.expression(exit=(T\_RBR, T\_END, T\_SEMICOLON))

if self.\_current.token == T\_RBR:

self.next()

return NodeExpression(term)

raise SyntaxErrorExpectedVSGot(T\_RBR, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().type)

elif self.\_current.type == TYPE\_OPERATOR:

raise SyntaxErrorExpectedVSGot(

(TYPE\_INT, TYPE\_ID, T\_LBR, T\_NOT,), self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().type

)

raise SyntaxErrorExpectedVSGot((TYPE\_INT, TYPE\_ID, T\_LBR, T\_NOT,), self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().type)

def binary\_expressions(self, expected: tuple, term\_function, exit\_token: tuple, term\_function\_kwargs={}, minus=False, precedence\_priority=-1):

first\_minus\_operator = None

if minus and self.\_current.token == T\_MINUS:

first\_minus\_operator = self.\_current

self.next()

left = term\_function(\*\*term\_function\_kwargs)

if first\_minus\_operator is not None:

left = NodeUnaryOperator(first\_minus\_operator, left)

if self.\_current.token in expected:

operator = self.\_current

self.next()

right = self.binary\_expressions(expected, term\_function, exit\_token, term\_function\_kwargs, precedence\_priority = precedence\_priority)

return NodeBinaryOperator(left, operator, right, precedence\_priority)

elif self.\_current.token in exit\_token:

return left

elif self.\_current.type in (TYPE\_INT, TYPE\_ID,):

raise SyntaxErrorExpectedVSGot([TYPE\_OPERATOR + " (кроме отрицания)"], self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().type)

elif self.\_current.token == T\_RBR:

raise SyntaxErrorExpectedVSGot([TYPE\_OPERATOR + " (кроме отрицания)"], self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().type,

custom\_message="\")\" ничего не закрывает")

raise SyntaxErrorExpectedVSGot([TYPE\_OPERATOR + " (кроме отрицания)"], self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().type)

def expression(self, exit=(T\_END, T\_SEMICOLON,)):

expression = self.binary\_expressions(

expected=(T\_PLUS, T\_MINUS),

term\_function=self.binary\_expressions,

exit\_token=exit,

minus=True,

precedence\_priority = 0,

term\_function\_kwargs={

"expected": (T\_MUL, T\_DIV),

"term\_function": self.binary\_expressions,

"exit\_token": (T\_PLUS, T\_MINUS) + exit,

"precedence\_priority": 1,

"term\_function\_kwargs": {

"expected": (T\_AND, T\_OR),

"term\_function": self.term,

"exit\_token": (T\_PLUS, T\_MINUS, T\_MUL, T\_DIV) + exit,

"precedence\_priority": 2

}

}

)

return NodeExpression(expression)

def var\_declaration(self, one\_exists = False):

if self.\_current.token == T\_ID:

element = self.\_current

self.next()

next\_element = self.var\_declaration(one\_exists = True)

return NodeVar(element, next\_element)

elif one\_exists:

return None

raise SyntaxErrorExpectedVSGot(T\_ID, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

def mark\_declaration(self, one\_exists = False):

expected = (T\_MARKSEND,)

if self.\_current.token == T\_INT:

element = self.\_current

self.next()

next\_element = self.mark\_declaration(one\_exists = True)

return NodeMark(element, next\_element)

elif one\_exists and self.\_current.token in expected:

self.next()

return None

if one\_exists:

expected = (T\_INT,) + expected

else:

expected = (T\_INT,)

raise SyntaxErrorExpectedVSGot(expected , self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

def mark(self, one\_exists = False):

expected = (T\_COLON,)

if self.\_current.token == T\_INT:

element = self.\_current

self.next()

next\_element = self.mark(one\_exists = True)

return NodeMark(element, next\_element)

elif one\_exists and self.\_current.token in expected:

return None

if one\_exists:

expected = (T\_INT,) + expected

else:

expected = (T\_INT,)

raise SyntaxErrorExpectedVSGot(expected , self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

def operation(self,number\_of\_operations=0) -> NodeOperation:

marks = self.mark()

if self.\_current.token == T\_COLON:

self.next()

if self.\_current.token == T\_ID:

variable = self.\_current

self.next()

if self.\_current.token == T\_EQ:

self.next()

expression = self.expression()

if self.\_current.token == T\_SEMICOLON:

self.next()

next\_expression = self.operation(number\_of\_operations + 1)

else:

next\_expression = None

return NodeOperation(variable, marks, expression, next\_expression)

else:

raise SyntaxErrorExpectedVSGot(T\_EQ, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

elif number\_of\_operations >= 1:

return None

raise SyntaxErrorExpectedVSGot(T\_ID, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

elif self.\_current.token == T\_END and number\_of\_operations >= 1:

return None

raise SyntaxErrorExpectedVSGot(T\_COLON, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

def program(self) -> NodeProgram:

self.next()

if self.\_current.token == T\_BEGIN:

self.next()

named\_set = self.named\_set()

operation = self.operation()

if self.\_current.token == T\_END:

self.next()

while self.\_current.token in SKIP\_TOKENS:

self.next()

if self.\_current.token == T\_EOF:

return NodeProgram(named\_set, operation)

else:

raise SyntaxErrorExpectedVSGot(T\_EOF, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value,

custom\_message="После \"Конец\" ничего не ожидалось", only\_custom=True)

else:

raise SyntaxErrorExpectedVSGot(T\_END, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().value)

print(self.\_current)

raise SyntaxErrorExpectedVSGot(T\_BEGIN, self.\_current.token, pos2d=self.\_current.pos2d, after=self.prev().token)

**Errors.py**

import tkinter as tk

from tokens import \*

def pos2d\_to\_message(pos2d):

return f" строка {pos2d[0][0]}"

class CustomSyntaxError(Exception):

def \_\_init\_\_(self, \*args: object) -> None:

super().\_\_init\_\_(\*args)

class SyntaxErrorExpectedVSGot(Exception):

def \_\_init\_\_(self, expected, got, pos2d=None, before=None,

exceptions=None, custom\_message=None, after=None,only\_custom=False) -> None:

assert isinstance(expected, (str, list, tuple))

if isinstance(expected, (list, tuple)):

expected\_str = f"[{' | '.join(expected)}]"

else:

expected\_str = expected

if isinstance(exceptions, (list, tuple)):

exceptions\_str = f"[{' | '.join(exceptions)}]"

else:

exceptions\_str = exceptions

self.message = "Ошибка:\n\t"

if custom\_message is not None:

self.message += custom\_message + "\n\t"

if not after in [TYPE\_INT, TYPE\_ID, TYPE\_OPERATOR, TYPE\_TERMINAL, T\_BOF, T\_EOF, T\_INT, T\_ID]:

after = f'"{after}"'

if not only\_custom:

self.message += f"После {after} не может следовать {got} \n\t(Должно быть {expected\_str})"

if exceptions is not None:

self.message += f" (кроме {exceptions\_str})"

if got == T\_EOF:

pos2d[0][0] -= 1

if pos2d is not None:

self.message += f"\nПоложение: \n\t{pos2d\_to\_message(pos2d)}"

self.pos2d = pos2d

super().\_\_init\_\_(self.message)

class SyntaxErrorMaxBracketsDepth(Exception):

def \_\_init\_\_(self, \*args: object) -> None:

super().\_\_init\_\_(\*args)

class TokenErrorUnrecognizedToken(Exception):

def \_\_init\_\_(self, last\_token, unrecognized\_token, cut=10) -> None:

self.token = last\_token

pos2d = self.token.pos2d

pos2d[1] = None

if pos2d is not None:

self.message = f"Ошибка: неопознанный токен: {unrecognized\_token[:-1]}...'\nПоложение: {pos2d\_to\_message(pos2d)}"

else:

self.message = f"Ошибка: неопознанный токен: {unrecognized\_token[:-1]}...'"

self.pos2d = [pos2d[1], None]

self.unrecognized\_token = unrecognized\_token

super().\_\_init\_\_(self.message)

**Nodes.py**

from tkinter import Variable

from tokens import \*

from utils import convert2serialize

import json

import math

class Node(object):

def \_\_init\_\_(self) -> None:

super().\_\_init\_\_()

def \_\_repr\_\_(self) -> str:

return super().\_\_repr\_\_()

def to\_json(self) -> str:

return json.dumps(convert2serialize(self))

class NodeSpaceSeparatedValues(Node):

def \_\_init\_\_(self, element, next\_element) -> None:

super().\_\_init\_\_()

self.element = element

self.next\_element = next\_element

def \_\_repr\_\_(self) -> str:

if self.next\_element is not None:

return f"{self.element} {self.next\_element}"

else:

return str(self.element)

class NodeBinaryOperator(Node):

def \_\_init\_\_(self, left, operator, right, precedence\_priority=-1) -> None:

super().\_\_init\_\_()

self.left = left

self.operator = operator

self.right = right

self.precedence\_priority = precedence\_priority

def \_\_repr\_\_(self) -> str:

return f"{self.left} {self.operator} {self.right}"

def compute(self):

flag = True

while flag:

if isinstance(self.right, NodeBinaryOperator):

if self.precedence\_priority == self.right.precedence\_priority:

flag = True

new\_left = NodeBinaryOperator(self.left, self.operator, self.right.left, self.precedence\_priority)

new\_operator = self.right.operator

new\_right = self.right.right

self.left, self.right, self.operator = new\_left, new\_right, new\_operator

else:

flag = False

else:

flag = False

left = self.left.compute()

right = self.right.compute()

return(self.compute\_binary\_operation(self.operator, left, right))

def update(self, variable, computation\_result):

self.left.update(variable, computation\_result)

self.right.update(variable, computation\_result)

@staticmethod

def compute\_binary\_operation(operator, left, right):

if left is not None and right is not None:

if operator.token == T\_MUL:

result = left \* right

elif operator.token == T\_DIV:

result = left / right

if operator.token == T\_PLUS:

result = left + right

elif operator.token == T\_MINUS:

result = left - right

elif operator.token == T\_AND:

result = left and right

elif operator.token == T\_OR:

result = left or right

return result

else:

return None

class NodeUnaryOperator(Node):

def \_\_init\_\_(self, operator, term) -> None:

super().\_\_init\_\_()

self.operator = operator

self.term = term

def \_\_repr\_\_(self) -> str:

return f"{self.operator}({self.term})"

def compute(self):

term = self.term.compute()

if term is not None:

if self.operator.token == T\_NOT:

result = not term

elif self.operator.token == T\_MINUS:

result = - term

return result

else:

return None

def update(self, variable, computation\_result):

self.term.update(variable, computation\_result)

class NodeExpression(Node):

def \_\_init\_\_(self, expression) -> None:

super().\_\_init\_\_()

self.expression = expression

def \_\_repr\_\_(self) -> str:

return f"({self.expression})"

def compute(self):

return self.expression.compute()

def update(self, variable, computation\_result):

self.expression.update(variable, computation\_result)

class NodeMark(NodeSpaceSeparatedValues):

def \_\_init\_\_(self, element, next\_element) -> None:

super().\_\_init\_\_(element, next\_element)

class NodeVar(NodeSpaceSeparatedValues):

def \_\_init\_\_(self, element, next\_element) -> None:

super().\_\_init\_\_(element, next\_element)

class NodeOperation(Node):

def \_\_init\_\_(self, variable, marks=None, expression=None, next\_operation=None, root\_operation=None) -> None:

super().\_\_init\_\_()

self.variable = variable

self.marks = marks

self.expression = expression

self.next\_operation = next\_operation

def \_\_repr\_\_(self) -> str:

s = f"{self.marks} : {self.variable} = {self.expression} "

if self.next\_operation is not None:

s += str(self.next\_operation)

return s

def compute(self):

result = self.expression.compute()

if result is not None:

whole, rem = str(float(result)).split(".")

whole = int(whole)

res = (oct(whole).replace("0o", "") or str(0)) + "."

print(result)

for x in range(8):

newDigit, rem = str(float(f"0.{rem}") \* 8).split(".")

res += str(newDigit) or str("0")

print(newDigit, rem, res)

message = f"Переменная {self.variable} = {res}\n"

self.variable.update(self.variable, result)

if self.next\_operation is not None:

self.next\_operation.update(self.variable, result)

message += self.next\_operation.compute()[1]

else:

message = f"Переменная {self.variable} - Ошибка, в связанных вычислениях используется неопределённая переменная"

return result, message

def update(self, variable, computation\_result):

self.expression.update(variable, computation\_result)

if self.next\_operation is not None:

self.next\_operation.update(variable, computation\_result)

class NodeNamedSet(Node):

def \_\_init\_\_(self, element, next\_named\_set) -> None:

super().\_\_init\_\_()

assert isinstance(

element,

(

NodeMark,

NodeVar

)

), f"Unsupported element type {type(element)} for NamedSet"

self.element = element

self.next\_named\_set = next\_named\_set

def \_\_repr\_\_(self) -> str:

if isinstance(self.element, NodeMark):

return f'"Метки" {self.element} {self.next\_named\_set} "Конец меток"'

elif isinstance(self.element, NodeVar):

return f'"Переменные" {self.element} {self.next\_named\_set}'

class NodeProgram(Node):

def \_\_init\_\_(

self,

named\_set: NodeNamedSet,

operations: NodeOperation

) -> None:

super().\_\_init\_\_()

self.named\_set = named\_set

self.operations = operations

def \_\_repr\_\_(self) -> str:

return f'"Начало" {self.named\_set} {self.operations} "Конец"'

**Tokenizer.py**

import re

from errors import TokenErrorUnrecognizedToken

from tokens import \*

from utils import \*

s2i = {"0": 0,

"1": 1,

"2": 2,

"3": 3,

"4": 4,

"5": 5,

"6": 6,

"7": 7,

}

class Tokenizer:

def \_\_init\_\_(self, code\_string):

self.\_code\_string = code\_string

self.\_pos = 0

def next\_token\_exists(self):

return self.\_pos < len(self.\_code\_string)

def tokenize(self):

self.tokens = [Token("skip", T\_BOF, "", [0, 0])]

try:

while self.next\_token\_exists():

token = self.next\_token()

if token is not None:

self.tokens.append(token)

self.tokens.append(Token("special", T\_EOF, "", pos=[self.\_pos, self.\_pos + 1]))

return self.tokens, None

except TokenErrorUnrecognizedToken as e:

return self.tokens, e

def next\_token(self):

current\_string = self.\_code\_string[self.\_pos:]

for pattern in PATTERNS + SKIP\_PATTERNS:

match = re.match(pattern.re, current\_string)

if match:

value = match.group(0)

token\_pos = [self.\_pos, self.\_pos + len(value)]

self.\_pos = token\_pos[1]

return Token(pattern.type, pattern.token, str(value), pos=token\_pos)

unrecognized\_token = repr(re.match(R'^.+s\*|(\n)', current\_string).group(0))

convert\_pos(self.tokens)

raise TokenErrorUnrecognizedToken(self.tokens[-1], unrecognized\_token)

**Tokens.py**

#########################################

# Tokens

# Words

T\_BEGIN = '"Начало"'

T\_END = '"Конец"'

T\_VARS = '"Переменные"'

T\_MARKS = '"Метки"'

T\_MARKSEND = '"Конец меток"'

# Values

T\_INT = 'Цел. Знач.'

T\_ID = 'Переменная'

# Operators

T\_PLUS = '"+"'

T\_MINUS ='"-"'

T\_MUL = '"\*"'

T\_DIV = '"/"'

T\_COMMA = '","'

T\_SEMICOLON = '";"'

T\_COLON = '":"'

T\_LBR = '"("'

T\_RBR = '")"'

T\_EQ = '"="'

T\_NOT = '"!"'

T\_AND = '"&"'

T\_OR = '"|"'

# Skip

TS\_SPACE = 'Пробел'

TS\_NEW\_LINE = '"Новая строка"'

TS\_TAB = '"Табуляция"'

# Other

T\_BOF = 'Начало документа'

T\_EOF = 'Конец документа'

#########################################

# Types

TYPE\_INT = 'Цел. Знач.'

TYPE\_ID = 'Переменная'

TYPE\_OPERATOR = 'Оператор'

TYPE\_TERMINAL = 'Терминал'

TYPE\_PLUS = '+'

TYPE\_MINUS ='-'

TYPE\_MUL = '\*'

TYPE\_DIV = '/'

TYPE\_COMMA = ','

TYPE\_LBR = '('

TYPE\_RBR = ')'

TYPE\_EQ = '='

TYPE\_POW = '^'

TYPE\_NOT = '!'

TYPE\_AND = '&'

TYPE\_OR = '|'

TYPE\_SEMICOLON = ';'

TYPE\_COLON = ':'

class Pattern(object):

def \_\_init\_\_(self, re, type, token) -> None:

super().\_\_init\_\_()

self.re = re

self.type = type

self.token = token

SKIP\_PATTERNS = [

Pattern(R"^\n", "skip", TS\_NEW\_LINE),

Pattern(R"^ ", "skip", TS\_SPACE),

Pattern(R"^\t+", "skip", TS\_TAB)

]

SKIP\_TOKENS = [

TS\_NEW\_LINE,

TS\_SPACE,

TS\_TAB

]

# TODO fix errors with marks, variables, integers and real values

PATTERNS = [

#########################################

# Other

# Pattern(R"^\d+:", "mark", T\_MARK),

#Pattern(R'^EXPR', "temp", T\_TEMP\_EXPR),

#########################################

# Words

Pattern(R'^Начало', TYPE\_TERMINAL, T\_BEGIN),

Pattern(R'^Переменные', TYPE\_TERMINAL, T\_VARS),

Pattern(R'^Метки', TYPE\_TERMINAL, T\_MARKS),

Pattern(R'^Конец меток', TYPE\_TERMINAL, T\_MARKSEND),

Pattern(R'^Конец', TYPE\_TERMINAL, T\_END),

#########################################

# Operands

Pattern(R"^[а-яА-Я][0-7]{0,3}", TYPE\_ID, T\_ID),

Pattern(R"^[0-7]+", TYPE\_INT, T\_INT),

#########################################

# Operators

# {"re": R"^(\+|\-|\\*|\/|\|\||&&|\[|\]|=|,)", "type": TYPE\_OPERATOR},

Pattern(R'^\+', TYPE\_PLUS, T\_PLUS),

Pattern(R'^\-', TYPE\_MINUS, T\_MINUS),

Pattern(R'^\\*', TYPE\_MUL, T\_MUL),

Pattern(R'^\/', TYPE\_DIV, T\_DIV),

Pattern(R'^\=', TYPE\_EQ, T\_EQ),

Pattern(R'^\,', TYPE\_COMMA, T\_COMMA),

Pattern(R'^\(', TYPE\_LBR, T\_LBR),

Pattern(R'^\)', TYPE\_RBR, T\_RBR),

Pattern(R'^\:', TYPE\_COLON, T\_COLON),

Pattern(R'^\;', TYPE\_SEMICOLON, T\_SEMICOLON),

Pattern(R'^\|', TYPE\_OR, T\_OR),

Pattern(R'^\&', TYPE\_AND, T\_AND),

]

class Token(object):

def \_\_init\_\_(self, type, token, value, pos=None):

self.type = type

self.token = token

self.value = value

self.pos = pos

self.pos2d = None

self.computation\_result = None

def \_\_repr\_\_(self) -> str:

return str(self.value)

def compute(self):

if self.token in (T\_INT,):

return int(str(self.value), base = 8)

elif self.token == T\_ID:

if self.computation\_result is not None:

return self.computation\_result

return None

else:

raise Exception(f"compute() не поддерживается для {self.token}")

def update(self, var, val):

if self.token == T\_ID and var.value == self.value:

self.computation\_result = val

**Window.py**

import tkinter as tk

from tkinter import messagebox

from tkinter import ttk

from code\_input import \*

from tokenizer import Tokenizer

from text\_colors import TOKEN\_TYPE\_COLORS, ERROR\_COLOR

from code\_parser import \*

import codecs

from utils import convert\_pos, pos2d\_to\_str

class Window(tk.Frame):

def \_\_init\_\_(self):

self.font = ("Consolas", 12)

self.window = tk.Tk()

self.window.title("BNF")

self.window.configure(bg='white')

self.window.geometry("1300x600")

self.window.resizable(0, 0)

self.menu()

self.code\_input\_field()

self.run\_result\_field()

self.bnf()

self.window.bind("<F5>", self.run\_translator)

def menu(self):

menu = tk.Menu(self.window)

self.window.config(menu=menu)

run\_menu = tk.Menu(menu, tearoff = 0)

menu.add\_cascade(label="Run", menu=run\_menu)

run\_menu.add\_command(label=" Run ", command=self.on\_btn\_run)

def code\_input\_field(self):

self.text = CustomText(self.window, width=80, font=self.font, selectbackground="#a8ccff")

self.vsb = tk.Scrollbar(self.window, orient="vertical", command=self.text.yview)

self.text.configure(yscrollcommand=self.vsb.set)

self.text.tag\_configure("bigfont", font=("Consolas", "20", "bold"))

self.linenumbers = TextLineNumbers(self.font, self.window, width=30)

self.linenumbers.attach(self.text)

self.linenumbers.grid(column=0, row=0, sticky="nsw")

self.text.grid(column=1, row=0)

self.vsb.grid(column=2, row=0, sticky="nse")

self.text.bind("<<Change>>", self.\_on\_change)

self.text.bind("<KeyRelease>", self.\_on\_text\_modified)

self.text.bind("<Configure>", self.\_on\_change)

for key in TOKEN\_TYPE\_COLORS:

if TOKEN\_TYPE\_COLORS[key] is not None:

self.text.tag\_config(key, background=TOKEN\_TYPE\_COLORS[key][0], foreground=TOKEN\_TYPE\_COLORS[key][1])

self.text.tag\_config("error", background=ERROR\_COLOR[0], foreground=ERROR\_COLOR[1])

def bnf(self):

self.bnf = tk.Text(self.window, width = 65)

self.bnf.grid(column=4, row=0, sticky="NE")

with codecs.open("bnf.txt", "r", "utf-8") as f:

self.bnf.insert(tk.END, f.read())

def \_on\_text\_modified(self, event):

code\_string = self.text.get("1.0", tk.END)

tokens, tokenizer\_error = Tokenizer(code\_string).tokenize()

if tokenizer\_error is None:

convert\_pos(tokens)

for t in tokens:

pos2d = pos2d\_to\_str(t.pos2d)

if t.type in TOKEN\_TYPE\_COLORS:

if TOKEN\_TYPE\_COLORS[t.type] is not None:

self.text.tag\_add(t.type, pos2d[0], pos2d[1])

else:

self.text.tag\_add("default", pos2d[0], pos2d[1])

def \_on\_change(self, event):

self.linenumbers.redraw()

def run\_result\_field(self):

self.result\_field = tk.Text(self.window, font=self.font)

self.result\_field.grid(column=0, row=1, columnspan=5, sticky="we")

def open\_file(self):

pass

def close\_file(self):

pass

def run\_translator(self, \*args):

self.text.tag\_remove("error", "0.0", "999.999")

self.result\_field.delete("1.0", tk.END)

code\_string = self.text.get("1.0", tk.END)

tokens, tokenizer\_error = Tokenizer(code\_string).tokenize()

if tokenizer\_error is not None:

\_token = tokenizer\_error.unrecognized\_token

errorStart = self.text.search(\_token.split('\'')[1], "1.0")

errorEnd = errorStart.split('.')[0] + '.' + str(int(errorStart.split('.')[1]) + len(\_token))

self.text.tag\_add("error", errorStart, errorEnd)

self.result\_field.insert(tk.END, tokenizer\_error.message)

else:

convert\_pos(tokens)

P = Parser(tokens)

root\_node, parser\_error = P.ast()

if parser\_error is not None:

if hasattr(parser\_error, 'pos2d'):

pos2d = pos2d\_to\_str(parser\_error.pos2d)

self.text.tag\_add("error", pos2d[0], pos2d[1])

self.result\_field.insert(tk.END, str(parser\_error))

else:

cursor\_pos = self.text.index(tk.INSERT)

self.text.delete("1.0", tk.END)

self.text.insert(tk.END, code\_string[:-1])

self.text.mark\_set("insert", cursor\_pos)

\_, message = root\_node.operations.compute()

self.result\_field.insert(tk.END, message)

with open("s.json", "w") as f:

f.write(root\_node.to\_json())

def on\_btn\_run(self):

self.run\_translator()

def mainloop(self):

self.window.mainloop()

def main():

window = Window()

window.mainloop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Utils.py**

from tokenizer import \*

from tokens import \*

def convert2serialize(obj):

if isinstance(obj, dict):

return { k: convert2serialize(v) for k, v in obj.items() }

elif hasattr(obj, "\_ast"):

return convert2serialize(obj.\_ast())

elif not isinstance(obj, str) and hasattr(obj, "\_\_iter\_\_"):

return [ convert2serialize(v) for v in obj ]

elif hasattr(obj, "\_\_dict\_\_"):

return {

k: convert2serialize(v)

for k, v in obj.\_\_dict\_\_.items()

if not callable(v) and not k.startswith('\_')

}

else:

return obj

def convert\_pos(tokens: list, first\_row=1, first\_col=0):

row\_counter = 0

col\_counter = 0

for t in tokens:

row = row\_counter + first\_row

col1 = col\_counter + first\_col

col2 = col1 + len(t.value)

t.pos2d = [[row, col1], [row, col2]]

col\_counter += len(t.value)

if t.token == TS\_NEW\_LINE:

row\_counter += 1

col\_counter = 0

def pos2d\_to\_str(pos2d):

return [f"{pos2d[0][0]}.{pos2d[0][1]}", f"{pos2d[1][0]}.{pos2d[1][1]}"]

**Text\_Colors.py**

from tokens import \*

TOKEN\_TYPE\_COLORS = {

"skip": None,

"word": [None, "#070b26"],

"operand": [None, "#070b26"],

"operator": [None, "#070b26"],

"id": [None, "#070b26"],

"default": [None, "#070b26"]

}

ERROR\_COLOR = [None, "red"]