Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**Отчёт**

**по лабораторной работе №2**

**Дисциплина: Методы разработки трансляторов**

**Тема: «Перевод исходной программы в обратную польскую запись»**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_К. А. Корнилов

Направление подготовки 02.03.02 Фундаментальная информатика и

информационные технологии

Направленность (профиль) Математическое и программное обеспечение

компьютерных технологий

Преподаватель

д-р техн. наук, проф. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю. М. Вишняков

**СОДЕРЖАНИЕ**

[1 Вариант задания 3](#_Toc103025612)

[2 Понятие обратной польской записи 3](#_Toc103025613)

[3 Алгоритм Дейкстры 4](#_Toc103025614)

[4 Перевод операторов цикла в ОПЗ 7](#_Toc103025615)

[5 Результаты экспериментов 9](#_Toc103025616)

[Приложение А Листинг программы и комментарии к нему 13](#_Toc103025617)

**1 Вариант задания**

Вариант задания представляет собой пару: входной язык и выходной язык (таблица 1).

Таблица 1 – Вариант задания

|  |  |  |
| --- | --- | --- |
| № варианта | Входной язык | Выходной язык |
| 47 | PHP | R |

Разработать программу для перевода закодированного текста исходной программы в обратную польскую запись.

Программа получает на входе файл – результат лексического анализа и строит обратную польскую запись исходной программы.

Отчет по работе должен содержать полное описание алгоритма Дейкстры: таблицу приоритетов операторов и операций, а также алгоритм работы со стеком. Листинг программы и комментарии к нему, пример.

**2 Понятие обратной польской записи**

Обратная польская запись (ОПЗ) – представляет собой одну из форм записи выражений и операторов, отличительной особенностью которой является расположение аргументов (операндов) перед операцией (оператором).

Например, выражение, записанное в обычной скобочной записи,

(a+d)/c+b\*(e+d),

в ОПЗ имеет следующее представление:

ad+c/bed+\*+.

Обратная польская запись получила широкое распространение благодаря своему основному преимуществу ОПЗ может быть вычислена за один просмотр цепочки слева направо, который часто называют проходом.

**3 Алгоритм Дейкстры**

Исследованию формальных способов преобразования арифметических и логических выражений в ОПЗ посвящены многочисленные исследования, однако в практике системного программирования наибольшее распространение получили способы преобразования на основе алгоритма Дейкстры.

Суть алгоритма Дейкстры можно представить следующим рисунком:

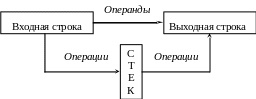


Рисунок 1 – Суть алгоритма Дейкстры

Из этого рисунка следует, что на вход алгоритма посимвольно поступает исходное выражение. Операнды исходного выражения пропускаются на выход и формируют так же посимвольно выходную строку. Операции обрабатываются по определенным правилам на основе стека.

Для реализации такой обработки известное в системном программировании понятие стека используется также в алгоритме Дейкстры для размещения в нем операций. При этом предварительно каждой операции приписывается свой приоритет на основе таблицы приоритетов, которая приведена ниже (таблица 2).

Таблица 2 – Таблица приоритетов

|  |  |
| --- | --- |
| Входной элемент | Приоритет |
| (  if  while  [  АЭМ  Ф  { | 0 |
| )  ,  ;  do  else  ] | 1 |
| = | 2 |
| Or  Xor | 3 |
| and | 4 |
| ! | 5 |

Продолжение таблицы 2

|  |  |
| --- | --- |
| Входной элемент | Приоритет |
| <  <=  !=  =  <>  >  >= | 6 |
| ^  &  |  ~  <<  >> | 7 |
| +  -  . | 8 |
| \*  /  % | 9 |
| \*\* | 10 |
| $ | 11 |
| }  function  return  echo | 12 |

**4 Перевод операторов цикла в ОПЗ**

Обработка оператора цикла с предусловием WHILE выражение DO оператор;:

1. Символ WHILE из входной строки заносится в стек в виде операции while:if.
2. Символ { выталкивает в выходную строку все операции из стека до ближайшего while:if. В выходную строку записывается while:if и в стек добавляется while:body.
3. Символ } указывает на конец оператора цикла с предусловием и выталкивает из стека все символы до ближайшего while:body, при этом сам WHILE уничтожается, а в выходную строку помещается операция while:body.

Обработка оператора цикла с постусловием while оператор do выражение; можно заменить последовательностью операторов

1. Символ do указывает на начало оператора цикла с постусловием и добавляет в стек символ do:st.
2. Символ while указывает на конец тела цикла и выталкивает из стека все символы до ближайшего do:st, добавляет в выходную последовательность символ do:body.
3. Символ ; указывает на конец условия работы цикла, выталкивает все содержимое стека и добавляет в выходную последовательность do:if.

**5 Результаты экспериментов**

Примеры работы программы для программы для тестирования.

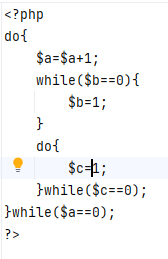


Рисунок 2 – Скриншот файла №1, содержащего текст на входном языке программирования



Рисунок 3 – Результат работы программы №1 в загруженном с первого этапа словаре.

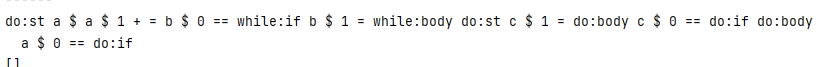


Рисунок 4 – Результат работы программы №1 в оригинальном языке.

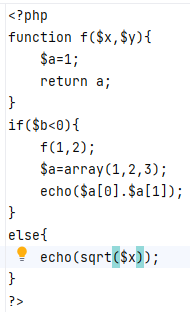


Рисунок 4 – Скриншот файла №2, содержащего текст на входном языке программирования



Рисунок 5 – Результат работы программы №2 в словаре с первого этапа.



Рисунок 6 – Результат работы программы №2 в оригинальном языке

**ПРИЛОЖЕНИЕ А   
Листинг программы и комментарии к нему**

import re  
  
from PyQt6.QtCore import QRunnable, QThread  
from PyQt6.QtGui import QTextCursor  
import json  
  
class LecAnalysis():  
 def createTokensCod(self,token\_class,token\_value):  
 if not(token\_value in self.tokens[token\_class]):  
 token\_code = str(len(self.tokens[token\_class])+1)  
 self.tokens[token\_class][token\_value] = token\_class + token\_code  
 def \_\_init\_\_(self):  
 self.OPERATIONS = ['$','.','+','-','\*','/','%','\*\*','=','==','!=','>','>=','<','<=','<>','^','&','|','~','<<','>>','!','and','or','xor']  
 self.CREATED\_FUNC = ['pow','sqrt','sin','cos','tan','abs','log','log10','max','min','array']  
 self.SERVICE\_WORDS = ['if','else','while','break','continue','function','return','echo','true','false','null','do','while:if','while:body',"do:st","do:body","do:if"]  
 self.SEPARATORS = [',',';','(',')',' ','\n','\t',"'",'"','<?','?>','{','}','[',']','#']  
 self.tokens = {'W':{},'I':{},'O':{},'R':{},'N':{},'C':{}}  
  
 *#Лаба 1* def process(self):  
 self.tokens = {'W': {}, 'I': {}, 'O': {}, 'R': {}, 'N': {}, 'C': {}}  
 for service\_word in self.SERVICE\_WORDS:  
 self.createTokensCod('W',service\_word)  
 for operation in self.OPERATIONS:  
 self.createTokensCod('O',operation)  
 for separator in self.SEPARATORS:  
 self.createTokensCod('R', separator)  
 for operation in self.CREATED\_FUNC:  
 self.createTokensCod('I',operation)  
 f = open('files/Input.txt','r')  
 input\_sequence = f.read()  
 f.close()  
  
  
 i=0  
 state = 'S'  
 output\_sequance = buffer = ''  
  
 add\_return\_counter=0  
 bracers\_func\_stack=[]  
 func\_buff=''  
 while i!=len(input\_sequence):  
 symbol = input\_sequence[i]  
 if (buffer == 'function' and func\_buff!='function'):  
 add\_return\_counter += 1  
 func\_buff='function'  
 elif (add\_return\_counter > 0 and buffer == 'return'):  
 add\_return\_counter -= 1  
 elif (symbol == '{'):  
 bracers\_func\_stack.append('{')  
 elif (symbol == '}'):  
 bracers\_func\_stack.pop()  
 if (add\_return\_counter > 0 and not bracers\_func\_stack):  
 add\_return\_counter -= 1  
 self.createTokensCod('C','')  
 output\_sequance += self.tokens['W']['return'] + ' '+ self.tokens['C']['']+ self.tokens['R'][';']  
 if(buffer!='function'):  
 func\_buff=''  
 if state == 'S':  
 buffer=''  
 if symbol=='$':  
 state = 'q9'  
 output\_sequance+=self.tokens['O']['$']+' '  
 elif symbol=='\_':  
 state='q26'  
 buffer=symbol  
 elif symbol=='?':  
 buffer = symbol  
 state='q28'  
 elif symbol.isalpha():  
 state = 'q1'  
 buffer=symbol  
 elif symbol.isdigit():  
 buffer=symbol  
 state = 'q3'  
 elif symbol=='.':  
 buffer=symbol  
 state = 'q7'  
 elif symbol == "'":  
 state = 'q8'  
 elif symbol =='"':  
 state='q10'  
 elif symbol == '/':  
 state = 'q15'  
 buffer=symbol  
 elif symbol == '#':  
 state = 'q19'  
 elif i == len(input\_sequence) - 1:  
 state = 'Z'  
 elif symbol in self.tokens['O'].keys():  
 state = 'q14'  
 buffer=symbol  
 elif symbol in self.tokens['R'].keys():  
 state='q20'  
 buffer=symbol  
 elif state == 'q9':  
 if symbol=='\_':  
 state='q26'  
 buffer=symbol  
 if symbol.isalpha():  
 state='q1'  
 buffer=buffer+symbol  
 elif symbol in self.tokens['R']:  
 state='S'  
 output\_sequance+=' '+self.tokens['R'][symbol]  
 buffer=''  
 elif state == 'q1':  
 if symbol.isalpha() or symbol=='\_':  
 buffer = buffer + symbol  
 state='q1'  
 elif symbol.isdigit():  
 buffer = buffer + symbol  
 state='q2'  
 else:  
 if buffer in self.SERVICE\_WORDS:  
 output\_sequance+=self.tokens['W'][buffer]+ ' '  
 elif buffer in self.OPERATIONS:  
 output\_sequance+=self.tokens['O'][buffer]+' '  
 else:  
 self.createTokensCod('I',buffer)  
 output\_sequance+=self.tokens['I'][buffer]+' '  
 state='S'  
 i=i-1  
 elif state == 'q2':  
 if symbol.isalpha() or symbol.isdigit() or symbol=='\_':  
 state='q2'  
 buffer = buffer + symbol  
 else:  
 if buffer in self.SERVICE\_WORDS:  
 output\_sequance+=self.tokens['W'][buffer]+ ' '  
 elif buffer in self.OPERATIONS:  
 output\_sequance+=self.tokens['O'][buffer]+' '  
 else:  
 self.createTokensCod('I',buffer)  
 output\_sequance+=self.tokens['I'][buffer]+' '  
 state='S'  
 i=i-1  
 elif state == 'q3':  
 if symbol.isdigit():  
 state='q3'  
 buffer = buffer + symbol  
 elif symbol=='.':  
 state='q4'  
 buffer = buffer + symbol  
 elif symbol=='E' or symbol=='e':  
 state='q5'  
 buffer = buffer + symbol  
 else:  
 self.createTokensCod('N', buffer)  
 output\_sequance += self.tokens['N'][buffer] + ' '  
 state='S'  
 i=i-1  
 elif state == 'q7':  
 if symbol.isdigit():  
 state='q4'  
 buffer = buffer + symbol  
 else:  
 i=i-1  
 output\_sequance +=self.tokens['O']['.']+' '  
 state='S'  
 elif state == 'q5':  
 if symbol.isdigit() or symbol=='-':  
 state='q6'  
 buffer = buffer + symbol  
 elif state == 'q6':  
 if symbol.isdigit():  
 state='q6'  
 buffer = buffer + symbol  
 else:  
 self.createTokensCod('N',buffer)  
 output\_sequance+=self.tokens['N'][buffer]+' '  
 state = 'S'  
 i=i-1  
 elif state == 'q4':  
 if symbol.isdigit():  
 state='q4'  
 buffer = buffer + symbol  
 else:  
 self.createTokensCod('N', buffer)  
 output\_sequance += self.tokens['N'][buffer] + ' '  
 state='S'  
 i=i-1  
 elif state == 'q8':  
 if symbol!="'":  
 state='q8'  
 buffer=buffer+symbol  
 else:  
 self.createTokensCod('C',buffer)  
 output\_sequance+=self.tokens['C'][buffer] + ' '  
 state = 'S'  
 *#НИКАКИХ ОБРАЩЕНИЙ К ЭЛЕМЕНТАМ МАССИВОВ В ЭТИХ СТРОКАХ и никаких множественных $* elif state == 'q10':  
 if symbol !='"' and symbol!="$":  
 state='q10'  
 buffer = buffer + symbol  
 elif symbol=='"':  
 self.createTokensCod('C', buffer)  
 output\_sequance += self.tokens['C'][buffer] + ' '  
 state = 'S'  
 else:  
 self.createTokensCod('C', buffer)  
 output\_sequance += self.tokens['C'][buffer] + ' ' + self.tokens['O']['.'] + ' ' + self.tokens['O']['$'] + ' '  
 state = 'q11'  
 buffer = ''  
 elif state == 'q11':  
 buffer=''  
 if symbol.isalpha():  
 state='q12'  
 buffer = buffer + symbol  
 elif symbol=='\_':  
 state='q27'  
 buffer = buffer + symbol  
 elif state == 'q12':  
 if symbol.isalpha() or symbol=='\_':  
 state='q12'  
 buffer = buffer + symbol  
 elif symbol.isdigit():  
 state='q13'  
 buffer = buffer + symbol  
 else:  
 if buffer in self.SERVICE\_WORDS:  
 output\_sequance+=self.tokens['W'][buffer]+ ' '  
 elif buffer in self.OPERATIONS:  
 output\_sequance+=self.tokens['O'][buffer]+' '  
 else:  
 self.createTokensCod('I',buffer)  
 output\_sequance+=self.tokens['I'][buffer]+ ' ' + self.tokens['O']['.'] + ' '  
 state='q10'  
 buffer=''  
 i=i-1  
 elif state == 'q13':  
 if symbol.isalpha() or symbol.isdigit() or symbol=='\_':  
 state='q13'  
 buffer = buffer + symbol  
 else:  
 if buffer in self.SERVICE\_WORDS:  
 output\_sequance+=self.tokens['W'][buffer]+ ' '  
 elif buffer in self.OPERATIONS:  
 output\_sequance+=self.tokens['O'][buffer]+' '  
 else:  
 self.createTokensCod('I',buffer)  
 output\_sequance+=self.tokens['I'][buffer]+' ' + self.tokens['O']['.'] + ' '  
 state='q10'  
 buffer = ''  
 i = i - 1  
 elif state == 'q14':  
 if symbol=='?':  
 buffer=buffer+symbol  
 self.createTokensCod('R',buffer)  
 state='S'  
 output\_sequance+=self.tokens['R'][buffer]+ ' '  
 elif symbol=='-':  
 state = 'S'  
 self.createTokensCod('O', buffer)  
 output\_sequance += self.tokens['O'][buffer] + ' '  
 i=i-1  
 elif symbol in [i[1] for i in self.tokens['O'] if len(i)==2]:  
 state='q14'  
 buffer=buffer+symbol  
 else:  
 state='S'  
 self.createTokensCod('O', buffer)  
 output\_sequance += self.tokens['O'][buffer]+ ' '  
 i=i-1  
 elif state == 'q15':  
 if symbol=='\*':  
 state='q16'  
 buffer=buffer+symbol  
 elif symbol=='/':  
 state='q17'  
 buffer=buffer+symbol  
 else:  
 state = 'S'  
 self.createTokensCod('O', buffer)  
 output\_sequance += self.tokens['O'][buffer] + ' '  
 i=i-1  
 elif state == 'q16':  
 if symbol=='\*':  
 state='q18'  
 buffer=''  
 else:  
 state='q16'  
 buffer=''  
 elif state == 'q17':  
 if i==len(input\_sequence)-1:  
 state='Z'  
 buffer=''  
 elif symbol=='\n':  
 buffer=''  
 state='S'  
 else:  
 buffer=''  
 state='q17'  
 elif state == 'q18':  
 if symbol!='/':  
 state='q16'  
 buffer=''  
 else:  
 state = 'S'  
 buffer=''  
 i=i+1  
 elif state == 'q19':  
 state='q17'  
 buffer=''  
 elif state == 'q20':  
 self.createTokensCod('R',buffer)  
 state='S'  
 if buffer =="\t":  
 output\_sequance+='\t'  
 elif buffer!=' ':  
 output\_sequance+=self.tokens['R'][buffer]+' '  
 if buffer=='\n':  
 output\_sequance += '\n'  
 i=i-1  
 elif state == 'q26':  
 if symbol=='\_':  
 state='q26'  
 buffer=buffer+symbol  
 else:  
 state='q1'  
 buffer=buffer+symbol  
 elif state == 'q27':  
 if symbol == '\_':  
 state = 'q27'  
 buffer = buffer + symbol  
 else:  
 state = 'q12'  
 buffer = buffer + symbol  
 elif state=='q28':  
 if symbol=='>':  
 buffer=buffer+symbol  
 self.createTokensCod('R', buffer)  
 state = 'Z'  
 output\_sequance += self.tokens['R'][buffer]  
 elif state=='Z':  
 break  
 i=i+1  
 for token\_class in self.tokens.keys():  
 with open('./files/%s.json' % token\_class, 'w') as write\_file:  
 data = {val: key for key, val in self.tokens[token\_class].items()}  
 json.dump(data, write\_file, indent=4, ensure\_ascii=False)  
 return output\_sequance,input\_sequence  
  
 *#Лаба 2* def get\_priority(self,token):  
 if token in ['(', 'if', 'while', '[', 'АЭМ', 'Ф', '{']:  
 return 0  
 if token in [')', ',', ';', 'do', 'else', ']',  
 ]:  
 return 1  
 if token == '=':  
 return 2  
 if token == 'or' or token=='xor':  
 return 3  
 if token == 'and':  
 return 4  
 if token == '!':  
 return 5  
 if token in ['<', '<=', '!=','<>', '==', '>', '>=']:  
 return 6  
 if token in ['^', '&', '|', '~', '<<', '>>']:  
 return 7  
 if token in ['+', '-', '.']:  
 return 8  
 if token in ['\*', '/', '%']:  
 return 9  
 if token in ['\*\*']:  
 return 10  
 if token in ['$']:  
 return 11  
 if token in ['}','function',  
 'return', 'echo',  
 ]:  
 return 12  
 return -1  
 def reverse\_polsk(self):  
 CLASSES\_OF\_TOKENS = ['W', 'I', 'O', 'R', 'N', 'C']  
  
 def is\_identifier(token):  
 return re.match(r'^I\d+$', inverse\_tokens[token])  
 self.tokens={}  
 *# файлы, содержащие все таблицы лексем* for token\_class in CLASSES\_OF\_TOKENS:  
 with open('./files/%s.json' % token\_class, 'r') as read\_file:  
 data = json.load(read\_file)  
 self.tokens.update(data)  
 *# лексемы (значение-код)* inverse\_tokens = {val: key for key, val in self.tokens.items()}  
 *# файл, содержащий последовательность кодов лексем входной программы* f = open('./files/Output.txt', 'r')  
 inp\_seq = f.read()  
 f.close()  
  
 regexp = '[' + '|'.join(CLASSES\_OF\_TOKENS) + ']' + '\d+'  
 match = re.findall(regexp, inp\_seq)  
  
 t = [self.tokens[i] for i in match]  
  
 i = 0  
 stack = []  
 out\_seq = ''  
 aem\_count = 2  
 proc\_level = operand\_count = 1  
 tag\_count = proc\_num = if\_count = while\_count = do\_count= \  
 begin\_count = end\_count = bracket\_count = 0  
 func\_count = 1  
 is\_if = is\_while = is\_do = is\_description\_var =False  
 is\_return = []  
  
  
 self.if\_marks=[]  
 self.else\_marks=[]  
 self.end\_marks=[]  
  
 self.while\_start\_marks=[]  
 self.while\_end\_marks=[]  
  
 self.do\_start\_marks=[]  
 self.do\_end\_marks=[]  
 while i < len(t):  
 print(out\_seq)  
 print(stack)  
 print(t[i])  
 print('------')  
 *# print(self.if\_marks)  
 # print(self.else\_marks)  
 # print(self.while\_start\_marks)  
 # print(self.while\_end\_marks)* p = self.get\_priority(t[i])  
 if p == -1:  
 if t[i]=='<?' and t[i+1]=='php':  
 out\_seq +=''  
 i=i+1  
 elif t[i]=='?>':  
 out\_seq+=''  
 elif t[i] != '\n' and t[i] != '\t':  
 out\_seq += t[i] + ' '  
 else:  
 if t[i] == '[':  
 if not re.match(r'^\d+АЭМ$', stack[-1]):  
 aem\_count = 2  
 stack.append(str(aem\_count) + 'АЭМ')  
 elif t[i] == ']':  
 while not (re.match(r'^\d+АЭМ$', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 if i < len(t) - 1 and t[i+1] == '[':  
 aem\_count = int(stack.pop().split('А')[0]) + 1  
 stack.append(str(aem\_count) + 'АЭМ')  
 i=i+1  
 else:  
 out\_seq += stack.pop() + ' '  
 aem\_count = 2  
 elif t[i] == '(':  
 if is\_identifier(t[i - 1]):  
 if t[i + 1] != ')':  
 func\_count = 1  
 stack.append(str(func\_count) + 'Ф')  
 else:  
 stack.append(t[i])  
 bracket\_count += 1  
 elif t[i] == ')':  
 while stack[-1] != '(' and not (re.match(r'^\d+Ф$', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 if re.match(r'^\d+Ф$', stack[-1]):  
 f\_c=int(stack[-1].split('Ф')[0])  
 if(t[i-1]!='('):  
 stack.append(str(f\_c+1) + 'Ф')  
 else:  
  
 stack.append(str(f\_c) + 'Ф')  
 out\_seq += stack.pop() + ' '  
 while len(stack) > 0 and stack[-1] != "1Ф":  
 stack.pop()  
 stack.pop()  
 stack.append('2')  
 func\_count = 1  
 stack.pop()  
 bracket\_count -= 1  
 if bracket\_count == 0:  
 if is\_if:  
 while stack[-1] != 'if':  
 out\_seq += stack.pop() + ' '  
 tag\_count += 1  
 stack[-1] += ' M' + str(tag\_count)  
 out\_seq += 'M' + str(tag\_count) + ' УПЛ '  
 is\_if = False  
 self.else\_marks.append('M' + str(tag\_count))  
 self.end\_marks.append('M' + str(tag\_count))  
 if is\_while:  
 while not (re.match(r'^while:if$', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 out\_seq+=stack.pop()+' '  
 stack.append('while:body')  
 is\_while = False  
 elif t[i] == ',':  
 while not (re.match(r'^\d+АЭМ$', stack[-1])) and \  
 not (re.match(r'^\d+Ф$', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 if re.match(r'^\d+АЭМ$', stack[-1]):  
 aem\_count += 1  
 stack.append(str(aem\_count) + 'АЭМ')  
 if re.match(r'^\d+Ф$', stack[-1]):  
 func\_count += 1  
 stack.append(str(func\_count) + 'Ф')  
 elif t[i] == 'if':  
 stack.append(t[i])  
 if\_count += 1  
 bracket\_count = 0  
 is\_if = True  
 elif t[i] == 'else':  
 while not (re.match(r'^if M\d+$', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 stack.pop()  
 tag\_count += 1  
 stack.append('if M' + str(tag\_count))  
 out\_seq += 'M' + str(tag\_count) + ' БП M' + str(tag\_count - 1) + ' : '  
 self.if\_marks.append('M' + str(tag\_count))  
 self.end\_marks.pop()  
 elif t[i] == 'while':  
 if not is\_do:  
 stack.append('while:if')  
 while\_count += 1  
 is\_while = True  
 else:  
 stack.append('do:if')  
 bracket\_count = 0  
 elif t[i] == 'do':  
 stack.append('do:body')  
 out\_seq += 'do:st '  
 do\_count += 1  
 bracket\_count = 0  
 elif t[i] == 'function':  
 proc\_num += 1  
 stack.append('function ' + str(proc\_num) + ' ' + str(proc\_level))  
 elif t[i]=='return':  
 is\_return.append('true')  
 elif t[i] == '{':  
 if len(stack) > 0 and re.match(r'^function', stack[-1]):  
 num = re.findall(r'\d+', stack[-1])  
 stack.pop()  
 out\_seq += '0Ф ' + str(num[0]) + ' ' + str(num[1]) + ' НП '  
 stack.append('function ' + str(proc\_num) + ' ' + str(proc\_level))  
 begin\_count += 1  
 proc\_level = begin\_count - end\_count + 1  
 stack.append(t[i])  
 elif t[i] == '}':  
 end\_count += 1  
 proc\_level = begin\_count - end\_count + 1  
 while stack[-1] != '{':  
 out\_seq += stack.pop() + ' '  
 stack.pop()  
 if len(stack) > 0 and re.match(r'^function', stack[-1]):  
 stack.pop()  
 if is\_return:  
 out\_seq += 'return КП '  
 is\_return.pop()  
 else:  
 out\_seq += 'КП '  
 if if\_count > 0 and re.match(r'^if M\d+$', stack[-1]):  
 tag = re.search('M\d+', stack[-1]).group(0)  
 j = i + 1  
 while j < len(t) and t[j] == '\n':  
 j += 1  
 if j >= len(t) or t[j] != 'else':  
 stack.pop()  
 out\_seq += tag + ' : '  
 if\_count -= 1  
 if do\_count > 0 and re.match(r'^do:body$', stack[-1]):  
 is\_do = True  
 out\_seq+='do:body '  
 stack.pop()  
 if while\_count > 0 and re.match(r'^while:body', stack[-1]):  
 stack.pop()  
 out\_seq += 'while:body '  
 while\_count -= 1  
 elif t[i] == ';':  
 if len(stack) > 0 and re.match(r'^function', stack[-1]):  
 num = re.findall(r'\d+', stack[-1])  
 stack.pop()  
 out\_seq += str(num[0]) + ' ' + str(num[1]) + ' НП '  
 elif len(stack) > 0 and stack[-1] == 'end':  
 stack.pop()  
 out\_seq += 'КП '  
 elif is\_description\_var:  
 proc\_num, proc\_level = re.findall('\d+', stack[-1])  
 stack.pop()  
 out\_seq += str(operand\_count) + ' ' + proc\_num + ' ' + proc\_level + \  
 ' КО '  
 is\_description\_var = False  
 elif if\_count > 0 or while\_count > 0 or do\_count>0:  
 while not (len(stack) > 0 and stack[-1] == '{') and \  
 not (if\_count > 0 and re.match(r'^if M\d+$', stack[-1])) and \  
 not (while\_count > 0 and re.match(r'^while:body', stack[-1]))\  
 and not (do\_count > 0 and re.match(r'^do:if', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 if if\_count > 0 and re.match(r'^if M\d+$', stack[-1]):  
 tag = re.search('M\d+', stack[-1]).group(0)  
 j = i + 1  
 while t[j] == '\n':  
 j += 1  
 if t[j] != 'else':  
 stack.pop()  
 out\_seq += tag + ' : '  
 if\_count -= 1  
 if while\_count > 0 and re.match(r'^while M\d+ M\d+$', stack[-1]):  
 tag = re.findall('M\d+', stack[-1])  
 out\_seq += tag[0] + ' БП ' + tag[1] + ' : '  
 while\_count -= 1  
 if do\_count > 0 and re.match(r'^do:if', stack[-1]):  
 stack.pop()  
 out\_seq += 'do:if '  
 do\_count -= 1  
 is\_do = False  
 else:  
 while len(stack) > 0 and stack[-1] != '{':  
 out\_seq += stack.pop() + ' '  
 else:  
 while len(stack) > 0 and self.get\_priority(stack[-1]) >= p:  
 out\_seq += stack.pop() + ' '  
 stack.append(t[i])  
 i += 1  
  
 while len(stack) > 0:  
 out\_seq += stack.pop() + ' '  
 out\_seq = re.sub(r'(\d)Ф', r'\1Ф', out\_seq)  
 out\_seq = out\_seq.replace(' 0Ф','')  
 print(out\_seq)  
 return ' '.join([inverse\_tokens[symbol] if symbol in inverse\_tokens.keys() else symbol for symbol in out\_seq.split(' ')])  
  
  
 *#Лаба 3* def translate\_to\_R(self):  
 CLASSES\_OF\_TOKENS = ['W', 'I', 'O', 'R', 'N', 'C']  
 self.tokens = {}  
 *# файлы, содержащие все таблицы лексем* for token\_class in CLASSES\_OF\_TOKENS:  
 with open('./files/%s.json' % token\_class, 'r') as read\_file:  
 data = json.load(read\_file)  
 self.tokens.update(data)  
 if token\_class == 'C':  
 for k in data.keys():  
 data[k] = re.sub(r"'([^']\*)'", r'"\1"', data[k])  
 self.tokens.update(data)  
 *# лексемы (значение-код)* inverse\_tokens = {val: key for key, val in self.tokens.items()}  
 replace = {'echo': 'print', '=': '<-', '!=': '!=',  
 '==': '==', '/': '/', '%': '%%','>':'>','>=':'>=','<':'<','<=':'<=',  
 '$':'','.':'paste(','+':'+','-':'-','\*':'\*','\*\*':'\*\*',  
 '^':'bitwXor(','&':'bitwAnd(','|':'bitwOr(','~':'bitwNot(','<<':'bitwShiftL(','>>':'bitwShiftR(',  
 'and':'&&','or':'||','xor':'xor(','!': '!',  
 'true':'TRUE','false':'FALSE','null':'NULL',  
 'do':'repeat','array':'c'  
 }  
 *# файл, содержащий обратную польскую запись* f = open('./files/reverse\_polsk.txt', 'r')  
 inp\_seq = f.read()  
 f.close()  
  
 t = re.findall(r'(?:\'[^\']\*\')|(?:"[^"]\*")|(?:[^ ]+)', inp\_seq)  
  
 def is\_identifier(token):  
 return ((token in inverse\_tokens) and re.match(r'^I\d+$', inverse\_tokens[token]))  
  
 def is\_constant(token):  
 return ((token in inverse\_tokens) and re.match(r'^C\d+$', inverse\_tokens[token])) or (  
 (token in inverse\_tokens) and re.match(r'^N\d+$',inverse\_tokens[token]))  
  
 def is\_operation(token):  
 return (token in inverse\_tokens) and re.match(r'^O\d+$', inverse\_tokens[token])  
  
 i = 0  
 stack = []  
 out\_seq = ''  
 is\_func = False  
 variable = 0  
  
 t=[self.tokens[i] if i in self.tokens.keys() else i for i in t]  
 tub\_num=0  
 markers = []  
  
 print(self.else\_marks)  
 print(self.if\_marks)  
 *# print(self.while\_start\_marks)  
 # print(self.while\_end\_marks)  
 # print(self.do\_start\_marks)  
 # print(self.do\_end\_marks)* while i < len(t):  
 *# print(out\_seq)* print(stack)  
 *# print(markers)  
 # print(t[i])  
 # print('-------')* if is\_func == True and not (is\_identifier(t[i])):  
 out\_seq += '{\n'  
 is\_func = False  
 if is\_identifier(t[i]) or is\_constant(t[i]):  
 if t[i] in inverse\_tokens and re.match(r'^C\d+$', inverse\_tokens[t[i]]):  
 stack.append(f'"{t[i]}"')  
 else:  
 stack.append(replace[t[i]] if t[i] in replace else t[i])  
 elif t[i] == 'НП':  
 stack.pop()  
 stack.pop()  
 func\_name = stack.pop()  
 out\_seq += '\t'\*tub\_num + func\_name.split('(')[0]+'=function('+func\_name.split('(')[1]  
 tub\_num = tub\_num + 1  
 is\_func = True  
 elif t[i] == 'КП':  
 tub\_num-=1  
 out\_seq += '\t'\*tub\_num+ '}\n'  
 elif t[i]=='return':  
 result = stack.pop()  
 out\_seq+='\t'\*tub\_num+'return('+result+');\n'  
 elif t[i] == 'УПЛ':  
 stack.pop()  
 arg1 = stack.pop()  
 out\_seq += '\t'\*tub\_num+f'if ({arg1})' + '{\n'  
 tub\_num += 1  
 elif t[i] == 'БП':  
 if (t[i - 1] in self.while\_start\_marks):  
 out\_seq += '\t'\*tub\_num+'}\n'  
 tub\_num -= 1  
 elif (t[i - 1] in self.if\_marks):  
 out\_seq += '\t'\*tub\_num+'}\nelse{\n'  
 out\_seq = out\_seq  
 elif t[i] == ':':  
 if(t[i-1] in self.end\_marks):  
 out\_seq += '\t' \* tub\_num + '}\n'  
 tub\_num -= 1  
 elif(t[i-1] in self.if\_marks):  
 out\_seq+='\t'\*tub\_num+'}\n'  
 tub\_num-=1  
 elif t[i]=='echo':  
 arg1=stack.pop()  
 out\_seq+='\t'\*tub\_num+f'print({arg1});\n'  
 elif is\_operation(t[i]):  
 if t[i]=='$':  
 out\_seq = out\_seq  
 elif t[i]=='.' and len(stack)>=2:  
 arg1=stack.pop()  
 arg2=stack.pop()  
 out\_seq += '\t'\*tub\_num+f'Rp{variable}='+f'paste({arg2},{arg1},sep="");\n'  
 stack.append(f'Rp{variable}')  
 variable+=1  
 elif t[i] == '=' and len(stack) >= 2:  
 arg1 = stack.pop()  
 arg2 = stack.pop()  
 out\_seq += '\t'\*tub\_num+ f'{arg2} <- {arg1};\n'  
 else:  
 operation = replace[t[i]] if t[i] in replace else t[i]  
 arg1 = stack.pop()  
 if t[i] != '!':  
 arg2 = stack.pop()  
 out\_seq+='\t'\*tub\_num+f'Rp{variable}='+f'({arg2} {operation} {arg1});\n'  
 stack.append(f'Rp{variable}')  
 variable+=1  
 else:  
 stack.append(f'({operation}{arg1})')  
 elif re.match("[0-9]+АЭМ",t[i]):  
 k = int(t[i].split('АЭМ')[0])  
 a = []  
 while k != 0:  
 a.append(stack.pop())  
 k -= 1  
 a.reverse()  
 stack.append('\t'\*tub\_num + a[0] + '[' + ','.join(a[1:]) + ']')  
 elif t[i] in ['break', 'continue']:  
 stack.append(replace[t[i]] if t[i] in replace else t[i])  
 arg0 = stack.pop()  
 out\_seq += '\t'\*tub\_num + f'\t{arg0};\n'  
 elif re.match(r"[0-9]+Ф",t[i]):  
 k = int(t[i].split('Ф')[0])  
 a = []  
 while k != 0:  
 a.append(stack.pop())  
 k -= 1  
 a.reverse()  
 if(i<len(t)-3 and t[i+3]!='НП'):  
 out\_seq+='\t'\*tub\_num+f"Rp{variable}="+a[0]+'(' + ', '.join(a[1:]) + ');\n'  
 stack.append(f"Rp{variable}")  
 variable+=1  
 else:  
 stack.append(a[0]+'(' + ', '.join(a[1:]) + ')')  
 elif t[i]=='while:if':  
 out\_seq+='\t'\*tub\_num+'while('+stack.pop()+'){\n'  
 tub\_num+=1  
 elif t[i]=='while:body':  
 tub\_num-=1  
 out\_seq+=tub\_num\*'\t'+'}\n'  
 elif t[i]=='do:st':  
 out\_seq+=tub\_num\*'\t'+'repeat {\n'  
 tub\_num += 1  
 elif t[i]=='do:body':  
 pass  
 elif t[i] == 'do:if':  
 out\_seq+=tub\_num\*'\t'+f"if({stack.pop()})break\n"+(tub\_num-1)\*'\t'+"}\n"  
 tub\_num-=1  
 else:  
 stack.append(t[i])  
 i += 1  
 stack.clear()  
 return out\_seq  
  
 *#Лаба 4* def error(self):  
 print(self.nxtsymb,self.row\_counter,22222)  
 out\_sq = 'Ошибка в строке '  
 f = open('./files/error.txt', 'a')  
 out\_sq += str(self.row\_counter)  
 f.write(out\_sq+'\n')  
 f.close()  
 return  
 def program(self):  
 self.scan()  
 if(self.nxtsymb!='<?'):  
 self.error()  
 self.scan()  
 if(self.nxtsymb!='php'):  
 self.error()  
 self.scan()  
 self.text()  
 if(self.nxtsymb!='?>'):  
 self.error()  
  
  
 def text(self):  
  
 self.no\_end\_op\_error = ['-','+','/','\*','\*\*',')',']','.','==','<=','>=','<','>'  
 ',','<>','!=','%',';','^','&','|','<<','>>','and','or','xor']  
 while(self.nxtsymb!='?>'):  
 if(self.nxtsymb=='function'):  
 self.procedure()  
 elif self.identifier() or self.nxtsymb=='echo':  
 self.scan()  
 if self.nxtsymb == '(':  
 self.brackets.append('(')  
 self.scan()  
 if self.nxtsymb != ')':  
 self.expression()  
 while self.nxtsymb == ',':  
 self.scan()  
 self.expression()  
 if self.nxtsymb != ')':  
 self.error()  
 if len(self.brackets)==0 or (self.brackets and self.brackets.pop() != '('):  
 self.error()  
 self.scan()  
 elif self.nxtsymb == '=':  
 self.scan()  
 self.expression()  
 if self.nxtsymb != ';':  
 self.row\_counter-=1  
 self.error()  
 self.row\_counter+=1  
 elif self.nxtsymb=='$':  
 self.scan()  
 if not self.identifier():  
 self.error()  
 elif self.nxtsymb=='return':  
 self.scan()  
 self.expression()  
 if self.nxtsymb!=';':  
 self.row\_counter-=1  
 self.error()  
 self.row\_counter+=1  
 elif self.nxtsymb == 'break':  
 self.break\_operator()  
 self.scan()  
 if self.nxtsymb != ';': self.error()  
 elif self.nxtsymb == 'continue':  
 self.continue\_operator()  
 self.scan()  
 if self.nxtsymb != ';': self.error()  
 elif self.nxtsymb=='if':  
 self.conditional\_operator()  
 elif self.nxtsymb == 'while':  
 self.while\_loop()  
 elif self.nxtsymb =='do':  
 self.do\_loop()  
 elif self.nxtsymb=='{':  
 self.compound\_operator()  
 elif self.nxtsymb=='[':  
 self.scan()  
 self.brackets.append('[')  
 self.expression()  
 if self.nxtsymb!=']':  
 self.error()  
 if len(self.brackets)==0 or (self.brackets and self.brackets.pop()!='['):  
 self.error()  
 self.scan()  
 if self.nxtsymb == '=':  
 self.scan()  
 self.expression()  
 if self.nxtsymb != ';': self.error()  
 elif self.nxtsymb==';':  
 self.scan()  
 else:  
 break  
  
 *# оператор break* def break\_operator(self):  
 return self.nxtsymb == 'break'  
  
 *# оператор continue* def continue\_operator(self):  
 return self.nxtsymb == 'continue'  
 *#Сканирование do-while* def do\_loop(self):  
 if self.nxtsymb !='do': self.error()  
 self.scan()  
 self.compound\_operator()  
 self.while\_loop()  
  
 *#Сканирование while* def while\_loop(self):  
 if self.nxtsymb != 'while': self.error()  
 self.scan()  
 if self.nxtsymb != '(': self.error()  
 self.brackets.append('(')  
 self.condition()  
 if self.nxtsymb != ')': self.error()  
 if len(self.brackets)==0 or (self.brackets and self.brackets.pop()!='('):  
 self.error()  
 self.scan()  
 *#Сканирование if* def conditional\_operator(self):  
 if self.nxtsymb != 'if': self.error()  
 self.scan()  
 if self.nxtsymb != '(': self.error()  
 self.condition()  
 self.brackets.append('(')  
 if self.nxtsymb != ')': self.error()  
 if len(self.brackets)==0 or (self.brackets and self.brackets.pop() != '('):  
 self.error()  
 self.scan()  
 self.text()  
 if self.nxtsymb == 'else':  
 self.scan()  
 self.text()  
  
 *# условие* def condition(self):  
 if self.unary\_log\_operation():  
 self.scan()  
 if self.nxtsymb != '(': self.error()  
 self.brackets.append('(')  
 self.log\_expression()  
 if self.nxtsymb != ')': self.error()  
 if len(self.brackets)==0 or (self.brackets and self.brackets.pop() != '('):  
 self.error()  
 self.scan()  
 else:  
 self.log\_expression()  
 while self.binary\_log\_operation():  
 self.log\_expression()  
  
 *# унарная логическая операция* def unary\_log\_operation(self):  
 return self.nxtsymb == '!'  
  
 *# логическое выражение* def log\_expression(self):  
 self.scan()  
 self.expression()  
 self.comparison\_operation()  
 self.scan()  
 self.expression()  
  
 *# операция сравнения* def comparison\_operation(self):  
 return self.nxtsymb in ['!=', '<', '<=', '==', '>', '>=','<>']  
  
 *# бинарная логическая операция* def binary\_log\_operation(self):  
 return self.nxtsymb == 'and' or self.nxtsymb == 'or' or self.nxtsymb=='xor'  
  
 *#Сканирование выражения* def expression(self):  
 if self.nxtsymb == '(':  
 self.brackets.append('(')  
 self.scan()  
 self.expression()  
 if self.nxtsymb != ')': self.error()  
 if len(self.brackets)==0 or (self.brackets and self.brackets.pop()!='('):  
 self.error()  
 self.scan()  
 elif self.nxtsymb =='$':  
 self.scan()  
 if not self.identifier():  
 self.error()  
 self.scan()  
 self.expression()  
 elif self.identifier():  
 self.scan()  
 if self.nxtsymb == '(':  
 self.brackets.append('(')  
 self.scan()  
 if self.nxtsymb != ')':  
 self.expression()  
 while self.nxtsymb == ',':  
 self.scan()  
 self.expression()  
 if self.nxtsymb != ')':  
 self.error()  
 if len(self.brackets) == 0 or (self.brackets and self.brackets.pop() != '('):  
 self.error()  
 self.scan()  
 elif self.nxtsymb == '[':  
 self.brackets.append('[')  
 self.scan()  
 self.expression()  
 if self.nxtsymb != ']': self.error()  
 if len(self.brackets) == 0 or (self.brackets and self.brackets.pop() != '['):  
 self.error()  
 self.scan()  
 elif self.number\_const() or self.constant():  
 self.scan()  
 else:  
 self.error()  
 if self.arithmetic\_operation():  
 self.scan()  
 self.expression()  
  
 *#Проверяем арифметическая ли операция* def arithmetic\_operation(self):  
 return self.nxtsymb in ['%', '\*','\*\*', '+', '-', '/','.','>>','>>','&','|','^']  
 *#Сканирование следующего символа* def scan(self):  
 self.i+=1  
 if(self.i>len(self.match)):  
 if self.nxtsymb not in ['\n',';','}']:  
 self.error()  
 else:  
 for token\_class in self.tokens.keys():  
 if self.match[self.i] in self.tokens[token\_class]:  
 self.nxtsymb = self.tokens[token\_class][self.match[self.i]]  
 self.printer()  
 if self.nxtsymb=='\n':  
 if self.match[self.i-1] in self.tokens['R'].keys() and self.tokens['R'][self.match[self.i-1]] not in ['{','}', ';','\n'] and self.i!=2:  
 self.error()  
 self.row\_counter += 1  
 self.scan()  
 *#Сканирование функций-процедуры* def procedure(self):  
 *#Сканируем название* self.scan()  
 if not self.identifier():  
 self.error()  
 self.scan()  
 *#Сканируем аргументы* if self.nxtsymb!='(':  
 self.error()  
 self.brackets.append('(')  
 self.scan()  
 if self.nxtsymb!=')':  
 self.list\_of\_names(type\_read=1)  
 if self.nxtsymb!=')':  
 print(self.nxtsymb,1)  
 self.error()  
 if len(self.brackets) == 0 or (self.brackets and self.brackets.pop() != '('):  
 print(self.brackets)  
 self.error()  
  
 self.scan()  
 if self.nxtsymb!='{':  
 self.error()  
 else:  
 self.compound\_operator()  
  
 *#Считываем список аргументов* def check\_is\_agrument(self,type\_read=0):  
 if(self.nxtsymb=='$'):  
 self.scan()  
 if not self.identifier():  
 self.error()  
 elif type\_read==1:  
 self.error()  
 else:  
 if not(self.constant() or self.number\_const()):  
 self.error()  
 def list\_of\_names(self,type\_read=0):  
 self.check\_is\_agrument(type\_read)  
 self.printer()  
 self.scan()  
 while(self.nxtsymb==','):  
 self.scan()  
 self.check\_is\_agrument(type\_read)  
 self.scan()  
 *#Проверка идентификатора* def identifier(self):  
 return self.nxtsymb in self.tokens['I'].values()  
 *#Проверка константы* def constant(self):  
 return self.nxtsymb in self.tokens['C'].values()  
 def number\_const(self):  
 return self.nxtsymb in self.tokens['N'].values()  
 *#Сканирование блока {}* def compound\_operator(self):  
 if self.nxtsymb != '{':  
 self.error()  
 self.brackets.append('{')  
 self.scan()  
 self.text()  
 if self.nxtsymb != '}':  
 self.error()  
 print(self.brackets)  
 if len(self.brackets) == 0 or (self.brackets and self.brackets.pop() != '{'):  
 self.error()  
  
 self.scan()  
  
 def analyzer(self):  
 self.i = -1  
 self.nxtsymb = None *# разбираемый символ* self.row\_counter = 1 *# счётчик строк  
  
 # лексемы* self.tokens = {'W': {}, 'I': {}, 'O': {}, 'R': {}, 'N': {}, 'C': {}}  
  
 *# файлы, содержащие все таблицы лексем* for token\_class in self.tokens.keys():  
 with open('./files/%s.json' % token\_class, 'r') as read\_file:  
 data = json.load(read\_file)  
 self.tokens[token\_class] = data  
  
 *# файл, содержащий последовательность кодов лексем входной программы* f = open('./files/Output.txt', 'r')  
 input\_sequence = f.read()  
 f.close()  
  
 regexp = '[' + '|'.join(self.tokens.keys()) + ']' + '\d+'  
 self.match = re.findall(regexp, input\_sequence)  
 print(self.tokens)  
 print(self.match)  
 f=open('./files/error.txt','w')  
 f.close()  
 self.brackets = []  
 self.program()  
  
 def printer(self):  
 print(self.nxtsymb,self.i,self.row\_counter)