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 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. Д. Гонтарев

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 |
| 34 | PHP | C++ |

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:

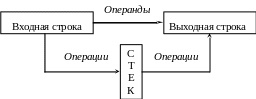


Рисунок 1 – Суть алгоритма Дейкстры

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 стеке к символу WHILE добавляется рабочая метка Mi и после этого в выходную строку записывается часть Mi:.
2. Символ DO выталкивает в выходную строку все операции из стека до ближайшего WHILE Mi. В стеке к WHILE Mi добавляется рабочая метка Mi+1 и после этого в выходную строку записывается часть Mi+1 УПЛ.
3. Символ ‘;’ указывает на конец оператора цикла с предусловием и выталкивает из стека все символы до ближайшего WHILE Mi Mi+1, при этом сам WHILE уничтожается, а в выходную строку помещается Mi БП Mi+1:.

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

1. Символ ‘;’ указывает на конец оператора цикла с постусловием и выталкивают из стека все символы до ближайшего do Mi, при этом сам do уничтожается, а в выходную строку помещаются Mi+1 УПЛ Mi БП Mi+1:.

**5 Результаты экспериментов**

Примеры работы программы для программы для тестирования.

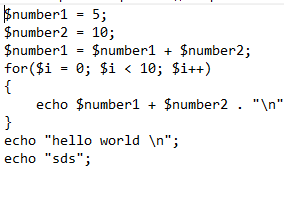


Рисунок 2 – Скриншот файла №1, содержащего текст на входном языке программирования



Рисунок 3 – Результат работы программы №1

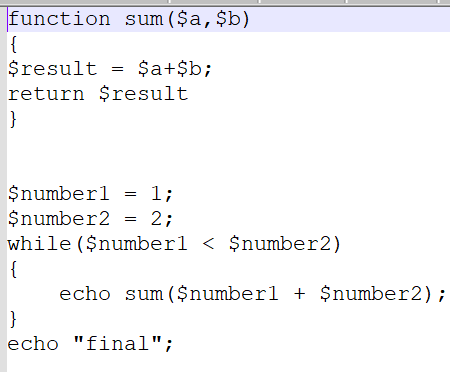


Рисунок 4 – Скриншот файла №2, содержащего текст на входном языке программирования



Рисунок 5 – Результат работы программы №2

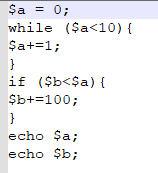


Рисунок 6 – Скриншот файла №3, содержащего текст на входном языке программирования



Рисунок 7 – Результат работы программы №3

**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']  
 self.SEPARATORS = [',',';','(',')',' ','\n','\t',"'",'"','<?','?>','{','}','[',']','#']  
 self.tokens = {'W':{},'I':{},'O':{},'R':{},'N':{},'C':{}}  
 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 = ''  
 while i!=len(input\_sequence):  
 symbol = input\_sequence[i]  
 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 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  
  
 *# self.SERVICE\_WORDS = [ 'break', 'continue',* 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  
 while i < len(t):  
 print(stack)  
 print(out\_seq)  
 print(t[i])  
 print(do\_count)  
 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])  
 stack.append(str(f\_c+1) + '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  
 if is\_while:  
 while not (re.match(r'^while M\d+$', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 tag\_count += 1  
 out\_seq += 'M' + str(tag\_count) + ' УПЛ '  
 stack[-1] += ' M' + str(tag\_count)  
 is\_while = False  
 elif t[i] == ',':  
 while not (re.match(r'^\d+АЭМ$', 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) + ' : '  
 elif t[i] == 'while':  
 if not is\_do:  
 tag\_count += 1  
 stack.append(t[i] + ' M' + str(tag\_count))  
 out\_seq += 'M' + str(tag\_count) + ' : '  
 while\_count += 1  
 is\_while = True  
 bracket\_count = 0  
 elif t[i] == 'do':  
 tag\_count += 1  
 stack.append(t[i] + ' M' + str(tag\_count))  
 out\_seq += 'M' + str(tag\_count) + ' : '  
 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] == '{':  
 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()  
 out\_seq += 'КП '  
 if if\_count > 0 and re.match(r'^if M\d+$', stack[-1]):  
 tag = re.search('M\d+', stack[-1]).group(0)  
 j = i + 1  
 while j < len(t) and t[j] == '\n':  
 j += 1  
 if j >= len(t) or t[j] != 'else':  
 stack.pop()  
 out\_seq += tag + ' : '  
 if\_count -= 1  
 if do\_count > 0 and re.match(r'^do M\d+$', stack[-1]):  
 is\_do = True  
 if while\_count > 0 and re.match(r'^while M\d+ M\d+$', stack[-1]):  
 tag = re.findall('M\d+', stack[-1])  
 stack.pop()  
 out\_seq += tag[0] + ' БП ' + tag[1] + ' : '  
 while\_count -= 1  
 elif t[i] == ';':  
 if len(stack) > 0 and re.match(r'^function', stack[-1]):  
 num = re.findall(r'\d+', stack[-1])  
 stack.pop()  
 out\_seq += str(num[0]) + ' ' + str(num[1]) + ' НП '  
 elif len(stack) > 0 and stack[-1] == 'end':  
 stack.pop()  
 out\_seq += 'КП '  
 elif is\_description\_var:  
 proc\_num, proc\_level = re.findall('\d+', stack[-1])  
 stack.pop()  
 out\_seq += str(operand\_count) + ' ' + proc\_num + ' ' + proc\_level + \  
 ' КО '  
 is\_description\_var = False  
 elif if\_count > 0 or while\_count > 0 or do\_count>0:  
 while not (len(stack) > 0 and stack[-1] == '{') and \  
 not (if\_count > 0 and re.match(r'^if M\d+$', stack[-1])) and \  
 not (while\_count > 0 and re.match(r'^while M\d+ M\d+$', stack[-1]))\  
 and not (do\_count > 0 and re.match(r'^do M\d+', stack[-1])):  
 out\_seq += stack.pop() + ' '  
 if if\_count > 0 and re.match(r'^if M\d+$', 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 M\d+', stack[-1]):  
 tag = re.findall('M\d+', stack[-1])  
 stack.pop()  
 tag\_count+=1  
 out\_seq += 'M'+str(tag\_count) + ' УПЛ ' + tag[0] + ' БП ' + 'M'+str(tag\_count) +' : '  
 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