**Федеральное агентство по образованию**Государственное образовательное учреждение высшего профессионального   
образования **«Тихоокеанский Государственный университет»**

Факультет компьютерных и фундаментальных наук

Кафедра ПОВТАС

**Лабораторная работа №2**

по дисциплине: «Теория и практика ЯП»

на тему: «Перевод исходной программы в обратную

польскую запись»  
Вариант №4

Выполнил: студент группы ПИИ(м)-21

Забавин А.С.

Проверил: кандидат физико-математических наук, доцент ка­федры ПОВТАС

Вихтенко Э.М.

# Постановка задачи

Необходимо перевести в обратную польскую запись код программы. Прежде разбитый лексическим анализатором на список лексем.

# Краткая теория

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:

a d + c / b e d + \* + .

Обратная польская запись получила широкое распространение

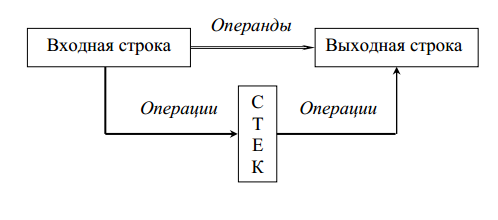
благодаря своему основному преимуществу ОПЗ может быть вычислена за

один просмотр цепочки слева направо, который часто называют проходом. Для эффективного разбора и оптимизации сканера, а также для его программной реализации в алфавите автомата выделяют несколько подмножеств символов, по которым сканер выполняет одинаковые переходы и действия.

# 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:



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.

# Результаты работы

В качестве исходного был выбран язык программирования C++. Лабараторная работа использует список лексем построенный в ЛР1 (см. Рис. 1).

Код программы представлен на листинге 1.

from collections import OrderedDict

from itertools import groupby

from shared\_classes import VarStack, VarQueue

from lab1\_lexical\_analysis.main import LexicalAnalyzerC

import time

class **DijkstratorC**():

\_cur\_op\_state = *'S'*

TABLE\_PRIORITY = {}

*@property*

def **TABLE\_PRIORITY\_ITEMS**(*self*):

itms = sorted(*self*.TABLE\_PRIORITY.items(), key=lambda x: x[1])

*self*.TABLE\_PRIORITY = dict(itms)

return itms

def **get\_by\_priority\_iterator**(*self*):

for prio, lexems in groupby(*self*.TABLE\_PRIORITY\_ITEMS, lambda x: x[1]):

for lxorder in lexems:

yield prio, *self*.la\_instance.get\_reg\_lexem\_by\_order(lxorder[0])

def **get\_priority\_by\_lexemid**(*self*, class\_code: str, lex\_id, lex\_text=None):

*"""*

*Таблица приоритетов Дисктры для языка*

*https://ru.wikipedia.org/wiki/Операторы\_в\_C\_и\_C++*

**:param** *class\_code: - код лексемы из методички*

**:param** *lex\_id: - идентификатор лексемы из таблицы 1 лаюораторной*

**:param** *lex\_text: - текст лексемы*

*"""*

tcstate = *self*.cur\_op\_state.startswith(*'TYPE\_CAST'*)

if tcstate:

return 14

elif lex\_text in (*'::'*):

return 16

elif lex\_text in (*'++'*, *'--'*, *'('*, *')'*, *'['*, *']'*, *'.'*, *'->'*, ):

return 15

elif lex\_text in (*'++~~~'*, *'--~~~'*, *'+'*, *'-'*, *'!'*, *'~'*, *'\*'*, *'&'*, *'new'*, *'new[]'*, *'delete '*, *'delete[]'*,):

return 14

elif lex\_text in (*'.\*'*, *'->\*'*):

return 13

elif lex\_text in (*'\*'*, *'/'*, *'%'*,):

return 12

elif lex\_text in (*'+'*, *'-'*,):

return 11

elif lex\_text in (*'<<'*, *'>>'*, ):

return 10

elif lex\_text in (*'<'*, *'<='*, *'>'*, *'>='*, ):

return 9

elif lex\_text in (*'=='*, *'!='*,):

return 8

elif lex\_text in (*'&'*, ):

return 7

elif lex\_text in (*'^'*, ):

return 6

elif lex\_text in (*'|'*, ):

return 5

elif lex\_text in (*'&&'*, ):

return 4

elif lex\_text in (*'||'*, ):

return 3

elif lex\_text in (*'?:'*, *'='*, *'+='*, *'-='*, *'\*='*, *'/='*,

*'%='*, *'<<='*, *'>>='*, *'&='*, *'|='*, *'^='*, *'throw'*,):

return 2

elif lex\_text in (*','*):

return 1

return 0

def **get\_priority\_by\_reg\_lex\_order**(*self*, order: int):

nexto = order + 1

nextlex = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto)

lx = *self*.la\_instance.get\_reg\_lexem\_by\_order(order)

# Если это префиксный декермент инкремент

if nextlex.class\_code in (*'I'*, *'N'*) and lx.text in (*'--'*, *'++'*):

return *self*.get\_priority\_by\_lexemid(lx.class\_code, lx.lex\_id, lx.text + *'~~~'*)

return *self*.get\_priority\_by\_lexemid(lx.class\_code, lx.lex\_id, lx.text)

def **\_\_init\_\_**(*self*, la\_instance: LexicalAnalyzerC):

*self*.la\_instance = la\_instance

if not len(la\_instance.ALL\_registry):

la\_instance.run\_analysis()

*self*.TABLE\_PRIORITY = {lx.order: *self*.get\_priority\_by\_lexemid(lx.class\_code, lx.lex\_id, lx.text) for lx in la\_instance.ALL\_registry}

*self*.\_lexem\_queue = VarQueue(la\_instance.ALL\_registry)

*self*.\_postfix\_stack = VarStack()

*self*.\_output\_list = []

*self*.\_cur\_op\_state = *'S'*

*self*.current\_state\_verbose = *''*

pass

STATES = [

*'S'*,

*'BLOCK\_BODY'*, # { BLOCK\_BODY }

*'FUNC\_BODY'*, # ( FUNC\_BODY )

*'TYPE\_CAST:S'*,

*'TYPE\_CAST:BLOCK\_BODY'*,

*'TYPE\_CAST:FUNC\_BODY'*,

]

def **stack\_top**(*self*):

return *self*.\_postfix\_stack.top()

def **stack\_pop\_and\_state**(*self*, state=*''*):

if state:

*self*.\_cur\_op\_state = state

return *self*.\_postfix\_stack.pop()

def **stack\_add\_and\_state**(*self*, lex, state=*''*):

*self*.\_postfix\_stack.add(lex)

if state:

*self*.\_cur\_op\_state = state

def **output\_and\_state**(*self*, lex, state=*''*):

*self*.\_output\_list.append(lex)

if state:

*self*.\_cur\_op\_state = state

*@property*

def **cur\_op\_state**(*self*):

return *self*.\_cur\_op\_state

*@cur\_op\_state.setter*

def **cur\_op\_state**(*self*, value):

state\_verbose = *''*

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

state\_verbose = value[1]

value = value[0]

if value not in *self*.STATES:

raise ValueError(*'Статус не из списка допустимых'*)

*self*.current\_state\_verbose = state\_verbose

*self*.\_cur\_op\_state = value

def **run\_postfix\_notatization**(*self*):

*self*.\_time\_start = time.time()

while not *self*.\_lexem\_queue.is\_empty:

cl = *self*.\_lexem\_queue.pop()

# Если лексема является числом, строкой, добавляем ее в ПОЛИЗ-массив

if cl.class\_code in (*'N'*, *'C'*, ):

*self*.output\_and\_state(cl)

# Если лексема идентификатор

elif cl.class\_code in (*'I'*):

nexto = cl.order + 1

nextlex = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto)

# Если лексема идентификатор переменной, добавляем ее в ПОЛИЗ-массив

if not nextlex or nextlex.text != *'('*:

*self*.output\_and\_state(cl)

# Если

elif nextlex and nextlex.text == *'('*:

next\_lex1 = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto + 1)

next\_lex12 = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto + 2)

# Если две следующие лексемы образуют "приеведение типа", то ставим статус 'TYPE\_CAST'

if next\_lex1 and next\_lex12 and next\_lex1.class\_code == *'W'* and next\_lex12.text == *')'*:

*self*.output\_and\_state(cl, *'TYPE\_CAST:'* + *self*.cur\_op\_state.replace(*'TYPE\_CAST:'*, *''*))

# Если лексема является символом функции, помещаем ее в стек

else:

*self*.stack\_add\_and\_state(cl, *'FUNC\_BODY'*)

# Если лексема является открывающей скобкой и сейчас TYPE\_CAST, пропустим лексему

elif *self*.cur\_op\_state.startswith(*'TYPE\_CAST'*) and cl.class\_code in (*'R'*) and cl.text == *'('*:

cl = *self*.\_lexem\_queue.pop()

# Если лексема является разделителем аргументов функции (например, запятая)

elif *self*.cur\_op\_state == *'FUNC\_BODY'* and cl.class\_code in (*'R'*) and cl.text == *','*:

#===============================================================================

# до тех пор, пока верхним элементом стека не станет

# открывающаяся скобка, выталкиваем элементы из стека в

# ПОЛИЗ-массив. Если открывающаяся скобка не встретилась,

# это означает, что в выражении либо неверно поставлен

# разделитель, либо несогласованы скобки

#===============================================================================

while *self*.stack\_top().text != *'('*:

cl = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(cl, *'FUNC\_BODY'*)

# Если лексема является операцией θ, тогда

elif cl.class\_code in (*'O'*, *'W'*):

nexto = cl.order + 1

nextlex = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto)

tcstate = *self*.cur\_op\_state.startswith(*'TYPE\_CAST'*)

op\_priority = *self*.get\_priority\_by\_reg\_lex\_order(cl.order)

# Если сейчас приведение типов

if tcstate:

is\_left\_op\_ = *self*.la\_instance.is\_left\_op(*'(type)'*)

is\_right\_op\_ = *self*.la\_instance.is\_left\_op(*'(type)'*)

# Если это префиксный декермент инкремент

elif nextlex.class\_code in (*'I'*, *'N'*) and cl.text in (*'--'*, *'++'*):

is\_left\_op\_ = *self*.la\_instance.is\_left\_op(cl.text + *'~~~'*)

is\_right\_op\_ = *self*.la\_instance.is\_right\_op(cl.text + *'~~~'*)

# Определяем левостороннесть и правостороннесть по стандартным методам

else:

is\_left\_op\_ = *self*.la\_instance.is\_left\_op(cl.text)

is\_right\_op\_ = *self*.la\_instance.is\_right\_op(cl.text)

# Если Служебное слово (с идущеми вслед числами, строками и переменными) - то считаем по-умолчанию ее правосторонней операцией как префиксные инкременты

if not is\_left\_op\_ and cl.class\_code == *'W'* and nextlex.class\_code in (*'I'*, *'N'*, *'C'*):

is\_right\_op\_ = True

#===============================================================================

# 1) пока приоритет θ меньше либо равен приоритету операции,

# находящейся на вершине стека (для лево-ассоциативных

# операций), или приоритет θ строго меньше приоритета

# операции, находящейся на вершине стека (для право-

# ассоциативных операций) выталкиваем верхние элементы

# стека в ПОЛИЗ-массив;

# 2) помещаем операцию θ в стек.

#===============================================================================

if is\_left\_op\_ and *self*.stack\_top():

while op\_priority <= *self*.get\_priority\_by\_reg\_lex\_order(*self*.stack\_top().order): # op\_priority <= self.get\_priority\_by\_reg\_lex\_order(self.stack\_top().order)

cl\_ = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(cl\_)

if cl.class\_code != *'W'* or *self*.stack\_top().class\_code in (*'W'*, *'O'*, *'I'*, *'N'*, *'C'*):

*self*.stack\_add\_and\_state(cl)

elif is\_right\_op\_ and *self*.stack\_top():

while op\_priority < *self*.get\_priority\_by\_reg\_lex\_order(*self*.stack\_top().order): # op\_priority < self.get\_priority\_by\_reg\_lex\_order(self.stack\_top().order)

cl\_ = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(cl\_)

if cl.class\_code != *'W'* or *self*.stack\_top().class\_code in (*'W'*, *'O'*, *'I'*, *'N'*, *'C'*):

*self*.stack\_add\_and\_state(cl)

tc, \_, cs = *self*.cur\_op\_state.rpartition(*':'*)

*self*.cur\_op\_state = cs

# Открывающую скобку всегда вносим в стек

elif cl.class\_code in (*'R'*) and cl.text == *'('*:

*self*.stack\_add\_and\_state(cl)

#===============================================================================

# Если лексема является закрывающей скобкой, выталкиваем

# элементы из стека в ПОЛИЗ-массив до тех пор, пока на вершине

# стека не окажется открывающая скобка. При этом открывающая

# скобка удаляется из стека, но в ПОЛИЗ-массив не добавляется.

# Если после этого шага на вершине стека оказывается символ

# функции, выталкиваем его в ПОЛИЗ-массив. Если в процессе

# выталкивания открывающей скобки не нашлось и стек пуст, это

# означает, что в выражении не согласованы скобки

#===============================================================================

elif cl.class\_code in (*'R'*) and cl.text == *')'*:

while *self*.stack\_top().text != *'('*:

cl = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(cl)

if *self*.stack\_top().text == *'('*:

cl = *self*.stack\_pop\_and\_state()

topl = *self*.stack\_top()

if topl:

nexto = topl.order + 1

nextlex = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto)

if *self*.cur\_op\_state == *'FUNC\_BODY'* and topl.class\_code == *'I'* and nextlex.text == *'('*:

topl = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(topl, *'BLOCK\_BODY'*)

# Открывающую {

elif cl.class\_code in (*'R'*) and cl.text == *'{'*:

*self*.stack\_add\_and\_state(cl)

#===============================================================================

# Если лексема является } скобкой, выталкиваем

# элементы из стека в ПОЛИЗ-массив до тех пор, пока на вершине

# стека не окажется { скобка. При этом {

# скобка удаляется из стека, но в ПОЛИЗ-массив не добавляется.

# Если в процессе выталкивания { скобки не нашлось и стек пуст, это

# означает, что в выражении не согласованы скобки

#===============================================================================

elif cl.class\_code in (*'R'*) and cl.text == *'}'*:

while *self*.stack\_top().text != *'{'*:

cl = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(cl)

if *self*.stack\_top().text == *'{'*:

cl = *self*.stack\_pop\_and\_state()

topl = *self*.stack\_top()

if topl:

nexto = topl.order + 1

nextlex = *self*.la\_instance.get\_reg\_lexem\_by\_order(nexto)

if *self*.cur\_op\_state == *'BLOCK\_BODY'* and topl.class\_code == *'I'* and nextlex.text == *'{'*:

topl = *self*.stack\_pop\_and\_state()

*self*.output\_and\_state(topl, *'S'*)

pass

*self*.\_time\_end = time.time()

*self*.cur\_op\_state = (*'S'*, *f'Приведение к обратной польской записи завершено. Время выполнения: {self.\_time\_end - self.\_time\_start} с.'*)

pass

def **\_\_str\_\_**(*self*):

return *' '*.join([x.text for x in *self*.\_output\_list]) + *'\n\n'* + *self*.current\_state\_verbose

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

opz = DijkstratorC(LexicalAnalyzerC(*r'input\_code.cpp'*))

opz.run\_postfix\_notatization()

with open(opz.la\_instance.program\_filename, *'rt'*, encoding=*'utf-8'*) as f:

print(f.read())

print(opz.la\_instance)

print(*'\nОбратная польская нотация\n'*)

print(opz)

pass

Листинг 2. Файл main.py – основной код программы

# Вывод

Программа была написана на ЯП Python 3. В качестве входного языка был выбран C++. В ходе программы код программы языка С++ преобразуется в набор лексем записанных в обратной польской записи. Полный код программы доступен на https://github.com/PaiNt-git/study.teor\_pl/tree/main/src/lab2\_postfix\_notation