Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

# Отчёт

по курсовой работе

Дисциплина: Объектно-ориентированное программирование

Тема: Reverse Polish Notation

Студент гр. 3331506/90401 Матвеев В. Д.

Преподаватель Ананьевский М. С.

Санкт-Петербург
2022

# ВВЕДЕНИЕ

Алгоритм обратной польской записи — алгоритм обработки строк. Разработан Чарльзом Хэмблином около 1950 года, на основе польской записи Яна Лукасевича. Широко применялась в советских калькуляторах до 1980х годов, поскольку производило вычисления за меньшее число команд, что позволяло сберечь немного программной памяти, с которой были проблемы.

Задача алгоритма: преобразовать исходную строку в инфиксной записи в обратную польскую, затем провести вычисление значения выражения.

# Описание алгоритма

В данной реализации алгоритма я использовал стек в качестве временного хранилища и строки в качестве постоянного для своих действий. Сам алгоритм можно разбить на несколько шагов.

**Первый** из них: рассматриваем поочередно каждый символ входной строки. Если этот символ - число, то просто помещаем его в выходную строку. Если знак операции (+, -, *, /, ^), то проверяем приоритет данной операции. Операция возведения в степень имеет наивысший приоритет (в моей реализации он равен 4). Операции умножения и деления – средний приоритет (3 в моем случае). Операции сложения и вычитания имеют меньший приоритет (равный 2). Наименьший приоритет (1) имеет открывающая скобка.

Данные приоритеты важны для дальнейшего рассмотрения. Встретив один из символов, не являющихся числом, необходимо проверить стек. Если стек все еще пуст, или находящиеся в нем символы (а находиться в нем могут только знаки операций и открывающая скобка, ведь числа сразу отправляются в выходную строку) имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.

Если символ, находящийся на вершине стека имеет приоритет, больший или равный приоритету текущего символа, то извлекаем символы из стека в выходную строку до тех пор, пока выполняется это условие.

Кроме того, остаются открывающие и закрывающие скобки. С открывающими все просто – они отправляются в стек (помните, у них приоритет 1?). А вот при встрече закрывающей скобки мы извлекаем в выходную строку все знаки до тех пор, пока не вытащим открывающую скобку, которую просто стираем.

**Второй** шаг наступает, когда вся строка пройдена. В этот момент необходимо убедиться, что стек, используемый нами для операций пуст. Если это так, все хорошо, можно приступать к третьему шагу. Если нет – необходимо достать все оставшиеся операции. Однако, доставать их нужно в

порядке приоритетности, чтобы под конец вычислений не закралась ошибка. Для этого опять проверяем приоритет оставшихся операций и вытаскиваем либо верхнюю из стека, либо следующую за ним, а верхнюю возвращаем обратно.

И наконец, **третий** шаг заключается в обычном вычислении значения выражения. Для этого нам нужен еще один стек для чисел. Пробегаем строку, полученную на втором шаге, складывая в стек числа, пока не встречаем операцию. При встрече операции – проводим ее над двумя верхними числами в стеки. Ответ помещается обратно в стек. И так, пока не закончатся операции. Под конец в стеке останется одно число, которое и будет ответом.

# Исследование алгоритма

Произведём замеры времени выполнения алгоритма от количества повторений прохода всего алгоритма. Будем постепенно увеличивать количество циклов повторения, чтобы получить результат.

Тест будет проходить на следующей строке:

1/(7-(1+1))*3-(2+(1+1))*1/(7-(2+1))*3-(2+(1+1))*(1/(7-(1+1)))^-3-((2+(1+1)))+1/(7-((1+1)))*3-((2+(1+1))).

Зависимость скорости алгоритма от количества повторений представлена в таблице 1 и на рисунке 1.

Таблица 1 – Исследование скорости алгоритма

| Количество повторений | Время выполнения алгоритма, с |
|---|---|
| 500 | 0,015 |
| 1000 | 0,031 |
| 2000 | 0,062 |
| 5000 | 0,157 |
| 7500 | 0,234 |
| 10000 | 0,313 |
| 12500 | 0,407 |
| 15000 | 0,486 |

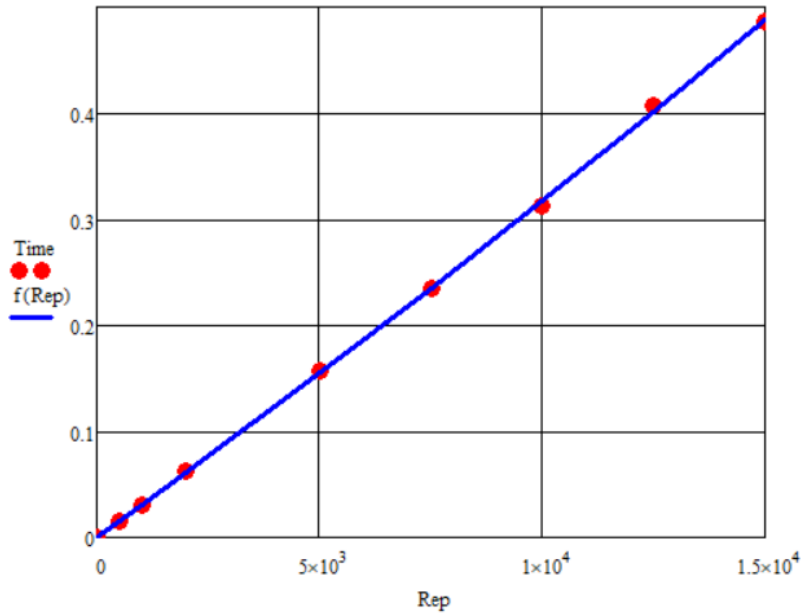


Рис. 5. График исследования затрачиваемого времени

Благодаря аппроксимации из графика видно, что зависимость времени воспроизведения алгоритма в зависимости от числа его повторений – линейна. Также, строка достаточно сложна, чтобы говорить о сохранении этой линейности при более простых и реальных строках.

# ЗАКЛЮЧЕНИЕ

В данном отчете был рассмотрен разработанный мной алгоритм обратной польской записи. Он позволяет брать привычные выражения, перебирать их, избавляться в от скобок и унарных операторов, чтобы затем представить их в виде постфиксной запись. Конечной задачей алгоритма является способность вычислить значение выражения, записанного в такой (постфиксной) записи.

Однако, есть несколько рекомендаций по применению этого алгоритма. Во-первых, достаточно простые строки (с небольшим количеством скобок и без унарных операторов) можно рассчитывать сразу же со второго метода моего класс – convert. Сложные же выражения, наоборот следует начинать с самого первого метода – он позволит проверить, все ли верно вы ввели, и правильно ли подготовился алгоритм (если нет – попробуйте добавить пару скобок в трудные места). Во-вторых, не пытайтесь рассчитывать значения выражений префиксной или инфиксной записи. Метод calculation рассчитан только на постфиксную запись.

## Список литературы

1. https://habr.com/ru/post/100869/ - описание алгоритма на примере поездов.

2. https://www.interface.ru/home.asp?artid=1492 – несколько предложений по реализации алгоритма.

3. https://www.youtube.com/watch?v=Z-i_RxB_TIg – видео-пример разработки алгоритма обратной польской записи. Язык – C#, НЕ C++.

# ПРИЛОЖЕНИЕ

Листинг 1 – Заголовочный файл алгоритма

```
#pragma once
#include <iostream>
#include <stack>
#include <string>
#include <cstring>
#include <stdlib.h>
#include <cmath>
#include <ctime>

//this class converts an expression to RPN
class Interpretation
{
public:
    std::string expression; //incoming equation
   std::string prepared; //equation after some magic)
    std::string rp_string; //equation in RPN

public:
   //Magic - the first pass along the string to get rid of the unary minus
   //You can start right from "convert", if your string dont have unary minus
   std::string prepare(std::string);
   //normal2PPN - conversion from usual math to polish math.
   //Just as was said you can start from this, if your string dont have unary
minus
    std::string convert(std::string);
   //Just calculation. But only for polish math))) Dont try to calculate
usual math here
    float calculate(std::string);

public:
   Interpretation();
   Interpretation(const Interpretation& other);
   Interpretation(Interpretation&& other) noexcept;
   ~Interpretation();

private:
   int input_id;//the index of the input string
   char actual_symbol;
   char prev_symbol;

private:
   static bool is_digit(char symb);//checking for numbers
   static int get_prior(char symb);
   char get_next_char(std::string);
   char get_prev_char(std::string);
};
```

Листинг 2 – Файл .cpp алгоритма

```
#include "RPN.h"

Interpretation::Interpretation()  = default;

Interpretation::Interpretation(const Interpretation & other)
{
   expression = other.expression;
   prepared = other.prepared;
   rp_string = other.rp_string;

   input_id = other.input_id;
   actual_symbol = other.actual_symbol;
   prev_symbol = other.prev_symbol;

}

Interpretation::Interpretation(Interpretation && other) noexcept
{
   expression = other.expression;
   prepared = other.prepared;
   rp_string = other.rp_string;
   input_id = other.input_id;
   actual_symbol = other.actual_symbol;
   prev_symbol = other.prev_symbol;

   other.input_id = 0;
   other.actual_symbol = '\0';
   other.prev_symbol = '\0';
   other.expression = "\0";
   other.prepared = "\0";
   other.rp_string = "\0";
}

Interpretation::~Interpretation() = default;

bool Interpretation::is_digit(char symb)
{
    return (symb >= '0' && symb <= '9');
}

int Interpretation::get_prior(char symb)
{
    switch (symb)
    {
        case '(':
            return 1;
        case '+':
        case '-':
            return 2;
        case '*':
        case '/':
            return 3;
        case '^':
            return 4;
        default:
            return 0;
    }
}

char Interpretation::get_next_char(std::string go_throw_str)
{
```

```
    if(input_id < go_throw_str.size())
    {
        return (actual_symbol = go_throw_str[input_id++]);
    }
    else
    {
        return (actual_symbol = '\0');
    }
}

char Interpretation::get_prev_char(std::string go_throw_str)
{
   return (prev_symbol = go_throw_str[input_id]);
}

std::string Interpretation::prepare(std::string expression)
{
   int prev_oper = 0;
   int bracket_num = 0;
   input_id = 0;
   while(get_next_char(expression) != '\0')
   {
      if(is_digit(actual_symbol))
      {
         prepared += actual_symbol;
         prev_oper = 0;
         if (get_prev_char(prepared) == '-')
         {
            prepared += ')';
            bracket_num--;
         }
         continue;
      }
      switch (actual_symbol)
      {
         case '+': case '*': case '/': case '^':
            prepared += actual_symbol;
            prev_oper = 1;
            continue;
         case '-':
            if (prev_oper == 0)
            {
               prepared += actual_symbol;
               prev_oper = 1;
               continue;
            }
            prepared += '(';
            bracket_num++;
            prepared += '0';
            prepared += actual_symbol;
            prev_oper = 1;
            continue;
         case '(':
            prepared += actual_symbol;
            bracket_num++;
            continue;
         case ')':
            while (bracket_num > 0)
            {
               prepared += actual_symbol;
               bracket_num--;
            }
            continue;
```

```
      }

   }

   return prepared;
}

std::string Interpretation::convert(std::string middle_str)
{
    int prev_oper = 0;
    int new_bracket = 0;
    input_id = 0;

    std::stack <char> stack4convert;
    std::stack <char> stack4operators;
    rp_string.clear();

    if((!is_digit(middle_str[0])) && middle_str[0] != '(')
    {
       rp_string = "";
      return ("Syntax error! Something wrong with the first symbol");
    }

    while(get_next_char(middle_str) != '\0')
    {
        if(is_digit(actual_symbol))
        {
            rp_string += actual_symbol;
            prev_oper = 0;
            continue;
        }

        switch (actual_symbol)
        {
            case '(':
                stack4convert.push(actual_symbol);
                new_bracket++;
                prev_oper = 0;
                break;

         case '*': case '/': case '+': case '-': case '^':
                if(input_id == middle_str.size())
                {
                   return ("Syntax error! Your last symbol - operator?!");
                }
                if(stack4convert.empty())
                {
                    prev_oper = 1;
                    stack4convert.push(actual_symbol);
                    break;
                }
            if((!prev_oper) && (!stack4convert.empty()))
                {
                    prev_oper = 1;
                    while(get_prior(actual_symbol) <=
get_prior(stack4convert.top()))
                    {
                        rp_string += stack4convert.top();
                        stack4convert.pop();
                        if (stack4convert.empty())
                        {
                            stack4convert.push(actual_symbol);
```

```
                            break;
                        }
                    }

                    if(get_prior(actual_symbol) >
get_prior(stack4convert.top()))
                    {
                        stack4convert.push(actual_symbol);
                    }
                    break;
                }
              return ("Syntax error! You got 2 operators going sequence.");
            case ')':
                if(prev_oper)
                {
                   return ("Syntax error! You got ')' after operator!");
                }
              while((actual_symbol = stack4convert.top())!= '(' &&
new_bracket>0)
              {
                 rp_string += actual_symbol;
                 stack4convert.pop();
              }
              new_bracket--;
            stack4convert.pop();
                break;
            default:
               return ("Error! Invalid symbol in the string.");
        }
    }

   char final_pop;
    /*while(!stack4convert.empty())
    {
       final_pop = stack4convert.top();
       stack4convert.pop();
       if(final_pop == '(')
       {
          continue;
       }
      stack4operators.push(final_pop);
    }*/
   while (!stack4convert.empty())
   {
      final_pop = stack4convert.top();
      stack4convert.pop();
      if(final_pop == '(')
      {
         continue;
      }
      if(stack4convert.empty())
      {
         rp_string += final_pop;
         continue;
      }
      if (get_prior(final_pop) >= get_prior(stack4convert.top()))
      {
         rp_string += final_pop;
         continue;
      }
      if (get_prior(final_pop) < stack4convert.top())
      {
         rp_string += stack4convert.top();
```

```
            stack4convert.pop();
            stack4convert.push(final_pop);
        }
    }

     if(new_bracket)
     {
        return ("Error. Wrong number of brackets.");
     }
    return rp_string;
}

float Interpretation::calculate(std::string rp_string)
{
    if((!is_digit(rp_string[0])) && rp_string[0] != '(')
    {
        rp_string = "";
        std::cout << "Calculation terminated! Something wrong with the first
symbol";
        return 0.0;
    }
    std::stack <float> stack4calculation;
    float f_num = 0.0;
    float s_num = 0.0;
    float temp = 0.0;
    input_id = 0.0;

    while(get_next_char(rp_string) != '\0')
    {
        if(is_digit(actual_symbol))
        {
            temp = float(actual_symbol - '0');
            stack4calculation.push(temp);
            continue;
        }
        else if(!is_digit(actual_symbol))
        {
            switch (actual_symbol)
            {
                case '+':
                    f_num = stack4calculation.top();
                    stack4calculation.pop();
                    s_num = stack4calculation.top();
                    stack4calculation.pop();
                    temp = f_num + s_num;
                    stack4calculation.push(temp);
                    break;
                case '-':
                    s_num = stack4calculation.top();
                    stack4calculation.pop();
                    f_num = stack4calculation.top();
                    stack4calculation.pop();
                    temp = f_num - s_num;
                    stack4calculation.push(temp);
                    break;
                case '*':
                    f_num = stack4calculation.top();
                    stack4calculation.pop();
                    s_num = stack4calculation.top();
                    stack4calculation.pop();
                    temp = f_num * s_num;
                    stack4calculation.push(temp);
                    break;
```

```
            case '/':
               s_num = stack4calculation.top();
               stack4calculation.pop();
               f_num = stack4calculation.top();
               stack4calculation.pop();
               if (s_num == 0)
               {
                  rp_string = "";
                  std::cout << "Calculation terminated! You've tried to
divide by 0";
                  return 0.0;
               }
               temp = f_num / s_num;
               stack4calculation.push(temp);
               break;
            case '^':
               s_num = stack4calculation.top();
               stack4calculation.pop();
               f_num = stack4calculation.top();
               stack4calculation.pop();
               temp = pow(f_num, s_num);
               stack4calculation.push(temp);
         }
      }
   }
   float answer = 0.0;
   answer = stack4calculation.top();
   stack4calculation.pop();
   return answer;
}
```