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

Курсовая работа

по дисциплине «Объектно-ориентированное программирование»

«Обратная польская запись»

Студент Арифуллина А.О.
Группа 3331506/20101
Преподаватель, доцент, к.т.н. Ананьевский М.С.

Санкт-Петербург
2025

Оглавление

Введение

В математике существует древняя традиция помещать оператор между операндами (x+y), а не после операндов (xy+). Форма с оператором между операндами называется инфиксной записью. Форма с оператором после операндов называется постфиксной, или обратной польской записью в честь польского логика Я. Лукасевича (1958), который изучал свойства этой записи.

Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул. Во-первых, любая формула может быть выражена без скобок. Во-вторых, она удобна для вычисления формул в машинах со стеками. В-третьих, инфиксные операторы имеют приоритеты, которые произвольны и нежелательны. Например, мы знаем, что ab+c значит (ab)+c, а не a(b+c), поскольку произвольно было определено, что умножение имеет приоритет над сложением. Но имеет ли приоритет сдвиг влево перед операцией И? Кто знает? Обратная польская запись позволяет устранить такие недоразумения.

Алгоритм ОПЗ широко используется в различных областях компьютерных наук и инженерии благодаря своей эффективности и удобству обработки математических выражений. Основные применения:

1. Вычислительные системы

- Калькуляторы (особенно научные и инженерные, например, HP, Casio)
- Интерпретаторы математических выражений (MATLAB, Wolfram Alpha)
- Финансовые вычисления (расчёт сложных формул в Excel-подобных системах)

2. Компиляторы и интерпретаторы

- Разбор арифметических выражений в языках программирования (например, в формулах a + b * c)
- Генерация промежуточного кода (например, байт-код в Java или .NET)

3. Компьютерная алгебра

- Символьные вычисления (анализ и упрощение формул в системах типа Mathematica)
- Решение уравнений (поиск корней, оптимизация)

4. Графические процессоры (GPU)

- Шейдерные вычисления (обработка сложных математических выражений в реальном времени)

5. Базы данных

- Оптимизация SQL-запросов с математическими условиями (например, WHERE (x + y) * 2 > 100)

Временная сложность:

Преобразование инфиксной → ОПЗ: Алгоритм Шейнинга-Ярда (Shunting-Yard) обрабатывает каждый токен ровно один раз, что даёт: O(n), где n — количество токенов во входной строке. Вычисление ОПЗ: Каждый токен

обрабатывается за O(1) (операции со стеком), поэтому общая сложность: O(n).

Итог: Общая временная сложность для полного преобразования и вычисления — O(n).

Пространственная сложность (память):

Стек операторов: В худшем случае (например, выражение с вложенными скобками (((...)))) стек может содержать O(n) операторов. Выходная строка (ОПЗ): Требует O(n) памяти.

Итог: Пространственная сложность — O(n).

Алгоритм

Алгоритм преобразования инфиксной записи в обратную польскую запись (ОПЗ) с учётом скобок и унарных операторов

Данная программа реализует алгоритм преобразования математических выражений из инфиксной формы в обратную польскую нотацию (ОПЗ) с последующим вычислением результата. Алгоритм состоит из двух основных этапов: преобразование выражения и его вычисление.

1. Преобразование инфиксной записи в ОПЗ

Алгоритм использует модифицированную версию алгоритма "Сортировочная станция" (Shunting-yard algorithm) Эдсгера Дейкстры. Основные шаги:

1) Инициализация:
   - Создается стек для операторов
   - Подготавливается поток для выходной строки
2) Обработка входной строки:
   - Посимвольный анализ строки с пропуском пробелов
   - Числа и десятичные дроби сразу добавляются в выходную строку
   - Для унарного минуса добавляется "0" перед числом
   - Открывающие скобки помещаются в стек
   - При обнаружении закрывающей скобки операторы извлекаются из стека до открывающей скобки
   - Операторы обрабатываются с учетом приоритетов: операторы с более высоким или равным приоритетом извлекаются из стека перед добавлением нового оператора
3) Завершающая обработка:
   - Оставшиеся в стеке операторы добавляются в выходную строку
   - Проверяется корректность скобок (отсутствие несбалансированных скобок)

Особое внимание уделяется обработке унарных операторов:

- Унарный минус определяется по контексту (начало выражения или после оператора/скобки)
- Для унарного минуса в выходную строку добавляется "0" перед операндом
- Унарный плюс игнорируется

Вычисление выражения в ОПЗ

Пример работы. Вход: "-3 + 4 * (2 - 1)"

Шаг	Токен	Действие	Стек	Выход
1	-	Унарный: добавить 0 → стек [-]	[-]	[0]
2	3	Добавить в выход	[-]	[0, 3]
3	+	Вытолкнуть - → добавить +	[+]	[0, 3, -]
4	4	Добавить в выход	[+]	[0, 3, -, 4]
5	*	Добавить * в стек (приоритет > +)	[+, *]	...
6	(	Добавить в стек	[+, *, ( ]	...
7	2	Добавить в выход	...	..., 2
8	-	Бинарный: добавить в стек	[+, *, (, -]	..., 2
9	1	Добавить в выход	...	..., 2, 1
10	)	Вытолкнуть - → удалить (	[+, *]	..., 2, 1, -
11	Конец	Вытолкнуть все операторы	[]	0 3 - 4 2 1 - * +

Пример вычисления выражения в обратной польской записи

Обратная польская запись идеально подходит для вычисления формул на компьютере со стеком. Формула состоит из n символов, каждый из которых является либо операндом, либо оператором. Алгоритм для вычисления формулы в обратной польской записи с использованием стека прост. Нужно просто прочитать обратную польскую запись слева направо. Если встречается операнд, его нужно пометить в стек. Если встречается оператор, нужно выполнить заданную им операцию.

В качестве примера рассмотрим вычисление следующего выражения: (8+2*5)/(1+3*2-4).

Шаг	Оставшаяся цепочка	Стек
1	8, 2,5, *+1,3,2*+4-/	8
2	2,5, *+1,3,2*+4-/	8,2
3	5*+1,3,2*+4-/	8,2,5
4	*+1,3,2*+4-/	8,10
5	+1,3,2*+4-/	18
6	1,3,2*+4-/	18,1
7	3,2*+4-/	18,1,3
8	2*+4-/	18,1,3,2
9	*+4-/	18,1,6
10	+4-/	18,7
11	4-/	18,7,4
12	-/	18,3
13	/	6

Заключение

В ходе выполнения курсовой работы был успешно реализован алгоритм Обратной польской записи. Можно заметить, что реализованный алгоритм не обрабатывает наличие пробелов в сами числах.

Практическая значимость работы заключается в возможности использования этого алгоритма в реальных задачах, таких как подсчеты, калькуляторы.

Источнкии

1. Dijkstra, E. W. (1961). «Shunting-yard algorithm».
2. Knuth, D. E. (1997). «The Art of Computer Programming». Vol. 1.

Приложение А

```
#include <iostream>
#include <stack>
#include <string>
#include <sstream>
#include <cctype>
#include <cmath>
#include <map>
#include <stdexcept>

using namespace std;

map<string, int> precedence = {
    {"^", 4},
    {"*", 3}, {"/", 3},
    {"+", 2}, {"-", 2},
    {"(", 1}
};
bool isOperator(const string& token) {
    return token == "+" || token == "-" || token == "*" || token == "/" || token == "^";
}
bool isUnary(const string& expr, size_t pos) {
    if (pos == 0) return true;
    char prev = expr[pos - 1];
    return prev == '(' || isOperator(string(1, prev));
}
double applyOperator(double a, double b, const string& op) {
    if (op == "+") return a + b;
    if (op == "-") return a - b;
    if (op == "*") return a * b;
    if (op == "/") {
        if (b == 0) throw runtime_error("Деление на ноль!");
        return a / b;
```

```
    }
    if (op == "^") return pow(a, b);
    throw runtime_error("Неизвестный оператор: " + op);
}
string infixToRPN(const string& infix) {
    stack<string> opStack;
    stringstream output;
    string token;
    for (size_t i = 0; i < infix.size(); ++i) {
        char c = infix[i];
        if (isspace(c)) continue;
        if (isdigit(c) || (c == '-' && isUnary(infix, i) && isdigit(infix[i+1]))) {
            if (c == '-') output << "0 ";
            while (i < infix.size() && (isdigit(infix[i]) || infix[i] == '.')) {
                output << infix[i++];}
            output << ' ';
            --i;
            continue;}
        token = string(1, c);
        if (token == "(") {
            opStack.push(token);}
        else if (token == ")") {
            while (!opStack.empty() && opStack.top() != "(") {
                output << opStack.top() << ' ';
                opStack.pop();}
            if (opStack.empty()) throw runtime_error("Несбалансированные скобки!");
            opStack.pop();}
        else if (isOperator(token)) {
            while (!opStack.empty() && precedence[opStack.top()] >= precedence[token]) {
                output << opStack.top() << ' ';
                opStack.pop();}
            opStack.push(token);}
        else {
```

```
            throw runtime_error("Некорректный символ: " + token);}
    }
    while (!opStack.empty()) {
        if (opStack.top() == "(") throw runtime_error("Несбалансированные скобки!");
        output << opStack.top() << ' ';
        opStack.pop();}
    return output.str();
}
double evaluateRPN(const string& rpn) {
    stack<double> s;
    istringstream iss(rpn);
    string token;
    while (iss >> token) {
        if (isOperator(token)) {
            if (s.size() < 2) throw runtime_error("Недостаточно операндов для " + token);
            double b = s.top(); s.pop();
            double a = s.top(); s.pop();
            s.push(applyOperator(a, b, token));
        }
        else {
            try {
                s.push(stod(token));
            } catch (...) {
                throw runtime_error("Некорректный токен: " + token);}}
}
    if (s.size() != 1) throw runtime_error("Неверное выражение");
    return s.top();
}
int main() {
    string infixExpr;
    cout << "Введите выражение (например, '3 + 4 * (2 - 1)'): ";
    getline(cin, infixExpr);
    try {
```

```
        string rpnExpr = infixToRPN(infixExpr);
        cout << "ОПЗ: " << rpnExpr << endl;
        double result = evaluateRPN(rpnExpr);
        cout << "Результат: " << result << endl;
    } catch (const exception& e) {
        cerr << "Ошибка: " << e.what() << endl;}
    return 0;
}
```