МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Отчет по лабораторной работе**

**«Арифметические операции с полиномами»**

**Выполнил:**

студент группы 382003-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лапин Д.А.

**Проверил:**

к.т.н., сотрудник каф. МОСТ ИИТММ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Арисова А. Н.

Нижний Новгород  
2022

**Содержание**

[**Содержание** 2](#_Toc100056167)

[Введение 3](#_Toc100056168)

[**1** Постановка задачи 4](#_Toc100056169)

[**2** Руководство программиста 5](#_Toc100056170)

[**2.1** Описание структуры программы 5](#_Toc100056171)

[**2.2** Описание структур данных 5](#_Toc100056172)

[**2.3** Описание алгоритмов 7](#_Toc100056173)

[**2.3.1.** Лексический и синтаксический анализ 7](#_Toc100056174)

[**2.3.2.** Добавление монома к полиному 8](#_Toc100056175)

[Заключение 9](#_Toc100056177)

[Список литературы 10](#_Toc100056178)

[**Приложение** 11](#_Toc100056179)

# Введение

В лабораторной работе рассматривается вопрос выполнения арифметических операций над полиномами с использованием линейной динамической структуры данных односвязный список.

Разработка приложения для работы с полиномами является практически значимой задачей, поскольку оно позволит облегчить, ускорить и оптимизировать многие математические подсчеты.

С целью реализации арифметической работы над полиномами в процессе выполнения данной лабораторной работы подробно описывается список как динамическая структура данных, а также рассматриваются и разбираются методы и алгоритмы решения основных задач (основанных на структуре данных список), вытекающих из вопроса вычислений, производимых над полиномами и мономами: лексический анализ и синтаксический анализ корректности введённых полиномов, приведение подобных слагаемых, сортировка мономов в составе полинома для более удобной работы с ним и т.д.

# Постановка задачи

В данной лабораторной работе необходимо разработать программу, выполняющую арифметические операции с полиномами, в которой должна быть реализованы:

Работа с мономами от трех переменных (x, y, z), чья степень находится в пределах от 0 до 9. Cтепень хранится в свернутом виде, а именно в виде трехзначного числа (разряд сотен отвечает за степень переменной x, разряд десятков за степень переменной y, соответственно разряд единиц за степень z). Коэффициентами полинома являются вещественные числа.

Примеры мономов: xyz, zyx, y, x^2y^3z^4, xyyxxzx.

Операций сложения, вычитания, умножения на константу, умножения двух полиномов.

Пользователь имеет право работать с полиномами за счёт выполнения трёх перечисленных операций (с контролем приведения подобных слагаемых, удаления монома с нулевым коэффициентом и сохранением ограничений на степень каждой переменной в составе монома). Операцию умножения полинома на константу можно представить, как умножение полинома на полином, где в качестве одного из аргументов на вход передана константа – частный случай полинома. Должна быть гарантирована корректность выполняемых действий.

# Руководство программиста

## Описание структуры программы

С учетом сформулированных выше тезисов к реализации целесообразной представляется следующая структура программы:

* TList - класс, реализующий операции над односвязным списком. С классом работают 2 вспомогательных класса: node - звено односвязного списка и iterator - указатель на звено односвязного списка, для наиболее эффективной и оптимальной работы со списком.
* Polinom –класс, отвечающий непосредственно за арифметические операции над полиномами. С классом работает дополнительный класс - monom, в котором описан набор инструментов для работы с мономами.

Работа класса Polinom основывается на структуре списка и представляет собой односвязный список из мономов, для каждого звена определён указатель на следующий моном. Первое звено – фиктивное, моном со степенью равной -1.

* Parser – класс, отвечающий за парсинг введенных данных(преобразование строчек в полиномы). С классом работает дополнительный класс - Lexeme, который служит для разбиения полиномов на элементарные составляющие.

Кроме того, есть модуль программ тестирования test\_list.cpp и test\_polynom.cpp.

Реализация классов находится в приложении.

## Описание структур данных

В данном отчёте используется такая динамическая структура данных как список. Прежде всего дадим определение структуре данных и динамической структуре данных.

Структура данных - есть модель данных в виде математической структуры

*,*

где – базисные множества, – отношения между элементами базисных множеств. [[1]](#footnote-1)

Динамическая структура есть математическая структура, которой соответствует частично-упорядоченное (по включению) базовое множество М, элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

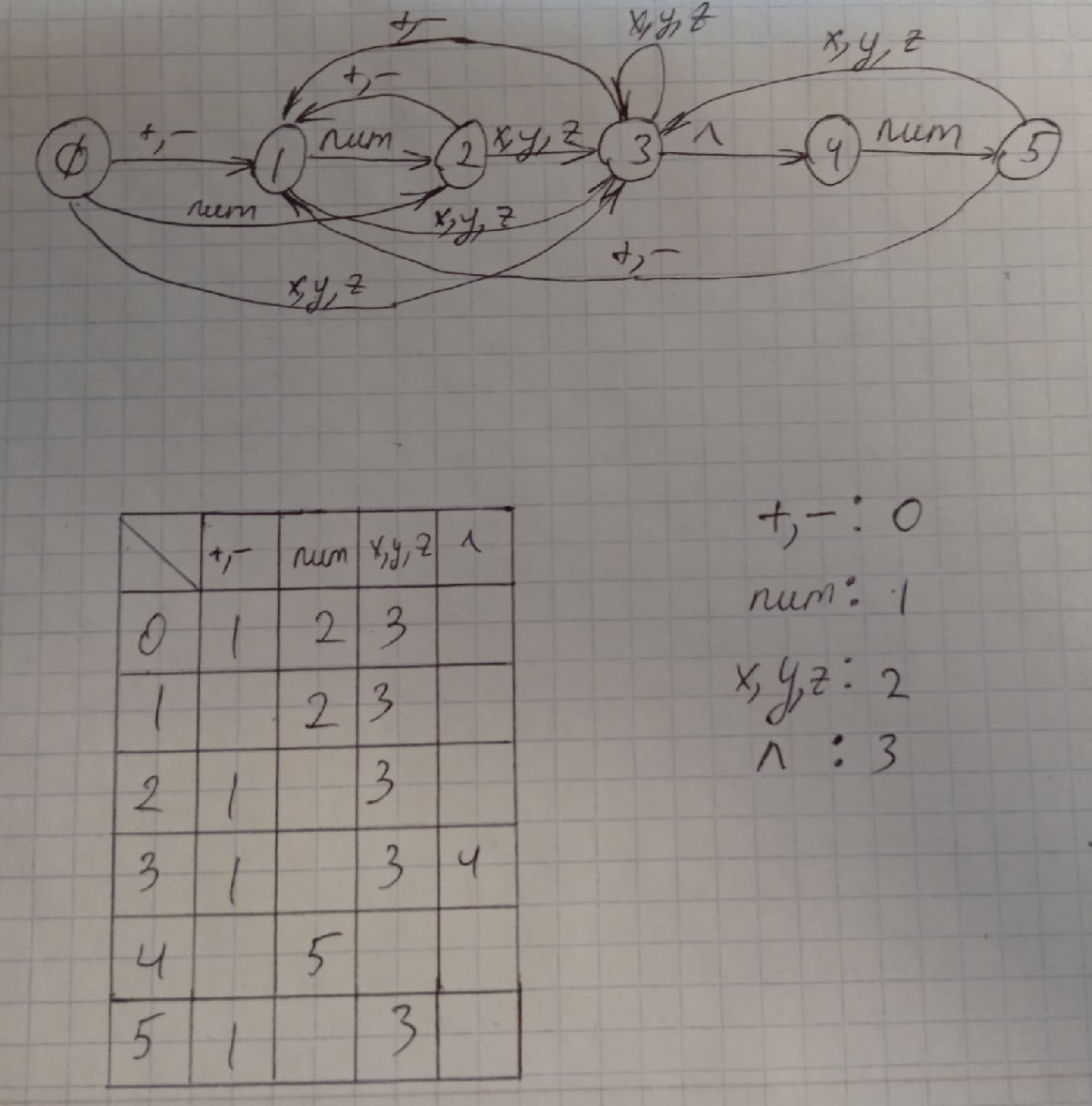
Связанный список (linked list) – это структура данных, в которой объекты расположены в линейном порядке. Однако в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на каждый объект. Связанные списки обеспечивают простое и гибкое представление динамических множеств.[1, c. 268].

В данном приложении используется односвязный список, в котором для каждого звена списка определён только указатель на следующее звено.

## Описание алгоритмов

### Лексический и синтаксический анализ

Алгоритм лексической и синтаксической проверки корректности введённой строки, задающей полином, а также разбиение его на мономы, реализует конструктор Parser(std::string input), описанная в приложении:

Мы сразу же получаем полином с помощью создания анонимного объекта Parser и вызова у него метода getPolinom(): Polinom pol = Parser(input).getPolinom(). В конструкторе происходят лексический разбор и синтаксический анализ. Лексический разбор происходит с помощью конечного автомата. При добавлении все мономы будут уже в отсортированном порядке (это мы увидим уже при анализе функции void add(Monom t). 

### Добавление монома к полиному

Для реализации данного алгоритма в функции void add(Monom t) используется проход в цикле по итератору.Add добавляет моном в полином, так, чтобы полином оставался отсортированным. Если в полиноме моном с такой же степенью, но отрицательным коэффициентом, то этот моном занулится и исчезнет из списка. Если передадим моном с нулевым коэффициентом, то он не добавится в полином. Сортировка мономов будет по трехзначному числу, представляющему собой степень переменных монома:

# Заключение

В ходе данной лабораторной работы было разработано приложение для работы с полиномами, которое поддерживает действия с десятичными и целыми коэффициентами мономов, операциями сложения, вычитания, умножения полинома на константу и перемножения двух полиномов. Были рассмотрены стандартные алгоритмы:

* синтаксическая проверка корректности введенного полинома,
* поддержка отсортированного состояния мономов, входящих в состав полинома,
* слияние двух упорядоченных списков (операция сложения полиномов) с дальнейшим приведением подобных слагаемых итогового полинома,

# Список литературы

1. Кормен Т. и др. Алгоритмы. Построение и анализ:[пер. с англ.]. – Издательский дом Вильямс, 2009.
2. Шилдт, Герберт. С++: базовый курс, 3-е изд. : Пер. с англ. – СПб.: ООО «Диалектика», 2019.-624 с.
3. Барышева И.В. и др. Под руководством Гергеля В.П. Учебно-методическое пособие. Нижний Новгород: Нижегородский университет, 2017.

**Приложение**

**TList.h**

#pragma once

#include<iostream>

template <class TElement>

struct Node {

TElement data;

struct Node\* next;

Node(TElement Data, Node\* Next) : data(Data), next(Next) {

}

};

template<class TElement>

class ListIterator : public std::iterator<std::input\_iterator\_tag, Node <TElement>>

{

private:

ListIterator(Node <TElement>\* p) :p(p) {}

Node <TElement>\* p;

public:

ListIterator(const ListIterator& it) : p(it.p) {}

bool operator!=(ListIterator const& other) const {

return p != other.p;

}

bool operator==(ListIterator const& other) const {

return p == other.p;

}//need for BOOST\_FOREACH

TElement& operator\*() const {

return p->data;

}

ListIterator next() const{

return ListIterator(p->next);

}

ListIterator& operator++() {

p = p->next;

return \*this;

}

ListIterator operator++(int) {

ListIterator t(\*this);

operator++();

return t;

}

template<class P>

friend class TList;

};

template <class TElement>

class TList

{

private:

unsigned int sizeOfList;

Node <TElement>\* first;

Node <TElement>\* last;

public:

TList() :sizeOfList(0), first(0), last(0) {

}

TList(TList const& a) {

sizeOfList = 0;

first = 0;

last = 0;

if (a.sizeOfList != 0) {

sizeOfList = a.sizeOfList;

Node <TElement>\* temp = a.first;

Node <TElement>\* t = new Node<TElement>(temp->data, 0);

first = t;

temp = temp->next;

while (temp != 0) {

t->next = new Node<TElement>(temp->data, 0);

t = t->next;

temp = temp->next;

}

temp = a.first;

while (temp -> next != 0) {

temp = temp->next;

}

last = temp;

}

}

TList& operator=(const TList& a) {

if (this != &a) {

clear();

sizeOfList = 0;

if (a.sizeOfList != 0) {

sizeOfList = a.sizeOfList;

Node <TElement>\* temp = a.first;

Node <TElement>\* t = new Node<TElement>(temp->data, 0);

first = t;

temp = temp->next;

while (temp != 0) {

t->next = new Node<TElement>(temp->data, 0);

t = t->next;

temp = temp->next;

}

}

}

return \*this;

}

~TList() {

clear();

}

void push\_top(const TElement element) {

Node<TElement>\* node = new Node <TElement>(element, first);

if (empty()) {

first = new Node <TElement>(element, first);

last = first;

}

else{

Node<TElement>\* t = new Node <TElement>(element, first);

first = t;

}

sizeOfList++;

}

void push\_back(const TElement element) {

Node<TElement>\* t = new Node<TElement>(element,0);

if(empty()){

first = t;

last = t;

}

else{

last->next = t;

last = t;

}

sizeOfList++;

}

void erase(ListIterator<TElement> t) {

if (t.p == 0) {

if (sizeOfList == 0) throw std::invalid\_argument("list is empty");

remove\_first();

}

Node<TElement>\* f = t.p->next;

t.p->next = t.p->next->next;

delete f;

sizeOfList--;

}

ListIterator<TElement> begin() const {

return ListIterator<TElement>(first);

}

ListIterator<TElement> end() const{

return ListIterator<TElement>(0);

}

void remove\_first() {

if (empty()) { throw std::invalid\_argument("list is empty"); }

if (first == last) {

last = 0;

}

Node<TElement>\* t = first;

first = first->next;

sizeOfList--;

delete t;

}

void remove\_last() {

if (empty()) { throw std::invalid\_argument("list is empty"); }

if (first == last) {

last = 0;

delete first;

first = 0;

sizeOfList--;

}

else {

Node<TElement>\* t = first;

while (t->next != last) t = t->next;

t->next = 0;

delete last;

last = t;

sizeOfList--;

}

}

void insert(ListIterator<TElement> t, const TElement& data) {

Node<TElement>\* f = t.p;

if (f==0) {

push\_top(data);

}

else {

f->next = new Node<TElement>(data, f->next);

sizeOfList++;

}

}

TElement& operator[] (const int index) {

if (index>=sizeOfList) throw std::invalid\_argument("invalid\_argument");

Node<TElement>\* p = first;

for (int i = 0; i < index; i++) {

p = p->next;

}

return p->data;

}

/\*void push(const TElement element) {

Node<TElement>\* node = new Node <TElement>(element, currentTop);

sizeOfStack++;

currentTop = node;

}\*/

void clear() {

while (sizeOfList)

pop();

}

/\*TElement top() {

return currentTop->data;

}\*/

void pop() {

if (sizeOfList == 0) throw std::invalid\_argument("pop out of empty list");

sizeOfList--;

Node<TElement>\* temp = first;

first = first->next;

delete temp;

}

int GetSize() {

return sizeOfList;

}

bool empty() {

return sizeOfList == 0;

}

friend std::ostream& operator<<(std::ostream& ostr, const TList& s) // вывод

{

Node<TElement>\* t = s.first;

while (t != 0) {

ostr << t->data << " ";

t = t->next;

}

return ostr;

}

};

**Polinom.h**#pragma once

#include "TList.h"

class Monom {

private:

double coeff;

int index;

public:

Monom(double coeff, int index):coeff(coeff),index(index)

{

if (!((index >= 0 && index <= 999) || index == -1)) throw std::invalid\_argument("invalid argument");

}

Monom operator-() const {

return Monom(-coeff, index);

}

friend std::ostream& operator<<(std::ostream& ostr, const Monom& s) // вывод

{

if (s.coeff == 0) ostr << "0";

else {

if (abs(s.coeff) != 1) ostr << s.coeff; else { if (s.coeff == -1) ostr << "-"; }

if (s.index / 100 != 0) { if (s.index / 100 != 1) ostr << "x^" << s.index / 100; else ostr << "x"; }

if (s.index % 100 / 10 != 0) { if (s.index % 100 / 10 != 1) ostr << "y^" << s.index % 100 / 10; else ostr << "y"; }

if (s.index % 10 != 0) { if (s.index % 10 != 1) ostr << "z^" << s.index % 10; else ostr << "z"; }

}

return ostr;

}

friend class Polinom;

friend inline bool operator< (const Monom& lhs, const Monom& rhs);

friend inline bool operator== (const Monom& lhs, const Monom& rhs);

friend inline bool operator== (const Monom& lhs, const Monom& rhs);

friend inline bool operator!= (const Monom& lhs, const Monom& rhs);

friend Monom operator+(Monom lhs, const Monom& rhs);

friend Monom operator\*(Monom lhs, const Monom& rhs);

friend Polinom operator+(const Polinom& lhs, const Polinom& rhs);

friend Polinom operator\*(const Polinom& lhs, const Polinom& rhs);

friend std::ostream& operator<<(std::ostream& ostr, const Polinom& s);

};

inline bool operator< (const Monom& lhs, const Monom& rhs) { return lhs.index < rhs.index; }

inline bool operator> (const Monom& lhs, const Monom& rhs) { return operator < (rhs, lhs); }

inline bool operator== (const Monom& lhs, const Monom& rhs) { return lhs.index == rhs.index; }

inline bool operator!= (const Monom& lhs, const Monom& rhs) { return lhs.index != rhs.index; }

Monom operator+(Monom lhs, const Monom& rhs) {

if (lhs.index != rhs.index) throw std::invalid\_argument("invalid\_argument");

lhs.coeff += rhs.coeff;

return lhs;

}

Monom operator\*(Monom lhs, const Monom& rhs) {

lhs.coeff \*= rhs.coeff;

lhs.index += rhs.index;

return lhs;

}

class Polinom {

private:

TList<Monom> pol;

public:

Polinom() {

pol.push\_back(Monom(0,-1));

}

Polinom(Polinom const& a){

pol=a.pol;

}

//максимально важная функция: add, добавляет моном в полином, так, чтобы полином оставался отсортированным.

//Если в полиноме моном с такой же степенью, но отрицательным коэффициентом, то этот моном занулится и исчезнет из списка

//если передадим моном с нулевым коэффициентом, то он не добавится в полином

void add(Monom t) {

if (t.coeff == 0) return;

auto it = pol.begin();

for (; it.next() != pol.end() && t < \*it.next(); it++){}

if (it.next() != pol.end()) {

if (t != \*(it.next())) {

pol.insert(it, t);

}

else {

(\*it.next()).coeff = (\*it.next()).coeff + t.coeff;

if ((\*it.next()).coeff == 0)

pol.erase(it);

}

}

else {

pol.insert(it, t);

}

}

Polinom operator-() const {

Polinom temp(\*this);

for (auto it = temp.pol.begin().next(); it != temp.pol.end(); it++)

\*it = -(\*it);

return temp;

}

friend std::ostream& operator<<(std::ostream& ostr,const Polinom& s) // вывод

{

if (s.pol.begin().next() != s.pol.end()) {

ostr << \*s.pol.begin().next();

for (auto it = s.pol.begin().next().next(); it != s.pol.end(); it++) {

if ((\*it).coeff > 0) {

ostr << "+" << \*it;

}

else {

ostr << \*it;

}

}

}

if(s.pol.begin().next() == s.pol.end()) {

ostr << "0";

}

return ostr;

}

Monom operator[] (const int index) {

return pol[index+1];

}

friend Polinom operator+(const Polinom& lhs, const Polinom& rhs);

friend Polinom operator\*(const Polinom& lhs, const Polinom& rhs);

};

Polinom operator+(const Polinom& lhs, const Polinom& rhs) {

Polinom temp;

auto lt = ++lhs.pol.begin();

auto rt = ++rhs.pol.begin();

while (lt != lhs.pol.end() && rt != rhs.pol.end()) {

if ((\*lt) > (\*rt)) {

temp.pol.push\_back(\*lt);

lt++;

}

else {

if ((\*lt) < (\*rt)) {

temp.pol.push\_back(\*rt);

rt++;

}

else {

int coeff = (\*lt).coeff + (\*rt).coeff;

if(coeff!=0){

temp.pol.push\_back(Monom(coeff, (\*lt).index));

lt++;

rt++;

}

else {

lt++;

rt++;

}

}

}

}

if (lt == lhs.pol.end()) {

for(;rt != rhs.pol.end();rt++)

temp.pol.push\_back(\*rt);

}

else {

for (; lt != lhs.pol.end(); lt++)

temp.pol.push\_back(\*lt);

}

return temp;

}

Polinom operator-(const Polinom& lhs, const Polinom& rhs) {

return (lhs+(-rhs));

}

Polinom operator\*(const Polinom& lhs, const Polinom& rhs) {

if (rhs.pol.begin().next() == rhs.pol.end() || lhs.pol.begin().next() == lhs.pol.end()) return Polinom();

int lt = (\*lhs.pol.begin().next()).index;

int rt = (\*rhs.pol.begin().next()).index;

if (((lt % 10) + (rt % 10) > 9) || (((lt % 100 / 10) + (rt % 100 / 10)) > 9) || (((lt / 100) + (rt / 100)) > 9)) throw std::invalid\_argument("multiply\_of\_polinoms\_out\_of\_range");

Polinom temp;

for (auto lt = lhs.pol.begin().next(); lt != lhs.pol.end(); lt++)

for (auto rt = rhs.pol.begin().next(); rt != rhs.pol.end(); rt++)

temp.add((\*rt) \* (\*lt));

return temp;

}

**Parser.h**

#pragma once

#include <iostream>

#include <vector>

#include <map>

#include<string>

#include "Polinom.h"

class Lexeme {

private:

std::string str;

int type;

public:

Lexeme(std::string str, int type):str(str), type(type) {

}

Lexeme(char str, int type) :str(std::string(1, str)), type(type) {

}

Lexeme(double str, int type) :str(std::to\_string(str)), type(type) {

}

int getType() {

return type;

}

friend std::ostream& operator<<(std::ostream& ostr, const Lexeme& s) // вывод

{

ostr << s.str;

return ostr;

}

friend class Parser;

};

class Parser {

std::vector<Lexeme> lex;

std::string input;

Polinom pol;

bool IsOperation(char op) {

if (op == '+' || op == '-' || op == '=')

return true;

else

return false;

}

bool IsVariable(char vr) {

if (vr == 'x' || vr == 'y' || vr == 'z')

return true;

else

return false;

}

bool IsAlpha(char ch) {

if ('0' <= ch && ch <= '9')

return true;

else

return false;

}

std::map <char, int> type = { {'+', 0},

{'-', 0},

{'0', 1}, {'^', 3},

{'x', 2}, {'y', 2}, {'z', 2}

};

//здесь и далее реализация конечного автомата

int pow\_x = 0;

int pow\_y = 0;

int pow\_z = 0;

double coeff = 0;

std::string current\_variable;

void reset() {

pow\_x = 0;

pow\_y = 0;

pow\_z = 0;

coeff = 0;

current\_variable = "";

}

//функции, которые пойдут в таблицу

void t\_0\_0(int i) {

if (lex[i].str == "+") coeff = 1;

else coeff = -1;

}

void t\_0\_1(int i) {

coeff = stod(lex[i].str);

}

void t\_0\_2(int i) {

current\_variable = lex[i].str;

coeff = 1;

if (lex[i + 1].str != "^") {

switch (lex[i].str[0]) {

case 'x': pow\_x = 1; break;

case 'y': pow\_y = 1; break;

case 'z': pow\_z = 1; break;

}

}

}

void t\_1\_1(int i) {

coeff = coeff \* stod(lex[i].str);

}

void t\_1\_2(int i) {

current\_variable = lex[i].str;

if (lex[i+1].str!="^") {

switch (lex[i].str[0]) {

case 'x': pow\_x = 1; break;

case 'y': pow\_y = 1; break;

case 'z': pow\_z = 1; break;

}

}

}

void t\_2\_0(int i) {

pol.add(Monom(coeff,0));

}

void t\_2\_2(int i) {

current\_variable = lex[i].str;

if (lex[i + 1].str != "^") {

switch (lex[i].str[0]) {

case 'x': pow\_x = 1; break;

case 'y': pow\_y = 1; break;

case 'z': pow\_z = 1; break;

}

}

}

void t\_3\_0(int i) {

if (!(pow\_x >= 0 && pow\_x < 10 && pow\_y >= 0 && pow\_y < 10 && pow\_z >= 0 && pow\_z < 10)) { throw std::invalid\_argument("pow of monom is out of range"); }

pol.add(Monom(coeff, 100 \* pow\_x + 10 \* pow\_y + pow\_z));

reset();

if (lex[i].str == "+") coeff = 1;

else coeff = -1;

}

void t\_3\_2(int i) {

current\_variable = lex[i].str;

if (lex[i + 1].str != "^") {

switch (lex[i].str[0]) {

case 'x': pow\_x += 1; break;

case 'y': pow\_y += 1; break;

case 'z': pow\_z += 1; break;

}

}

}

void t\_3\_3(int i) {

}

void t\_4\_1(int i) {

switch (current\_variable[0]) {

case 'x': pow\_x += (int)stod(lex[i].str); break;

case 'y': pow\_y += (int)stod(lex[i].str); break;

case 'z': pow\_z += (int)stod(lex[i].str); break;

}

}

void t\_5\_0(int i) {

if (!(pow\_x >= 0 && pow\_x < 10 && pow\_y >= 0 && pow\_y < 10 && pow\_z >= 0 && pow\_z < 10)) { throw std::invalid\_argument("pow of monom is out of range"); }

pol.add(Monom(coeff, 100\*pow\_x+10\*pow\_y+pow\_z));

reset();

if (lex[i].str == "+") coeff = 1;

else coeff = -1;

}

void t\_5\_2(int i) {

current\_variable = lex[i].str;

if (lex[i + 1].str != "^") {

switch (lex[i].str[0]) {

case 'x': pow\_x += 1; break;

case 'y': pow\_y += 1; break;

case 'z': pow\_z += 1; break;

}

}

}

//синтаксический анализ

void syntaxis\_analis() {

std::string tmp\_str = input;

while (tmp\_str[0] != '\0') {

char ch = tmp\_str[0];

if (IsOperation(ch)) {

lex.push\_back(Lexeme(ch, type[ch]));

tmp\_str.erase(tmp\_str.begin(), tmp\_str.begin() + 1);

}

else if (IsAlpha(ch)) {

std::string::size\_type sz;

lex.push\_back(Lexeme(std::stod(tmp\_str, &sz), type['0']));

tmp\_str.erase(tmp\_str.begin(), tmp\_str.begin() + sz);

}

else if (IsVariable(ch)) {

lex.push\_back(Lexeme(ch, type[ch]));

tmp\_str.erase(tmp\_str.begin(), tmp\_str.begin() + 1);

}

else if (tmp\_str[0] == '^') {

lex.push\_back(Lexeme(ch, type[ch]));

tmp\_str.erase(tmp\_str.begin(), tmp\_str.begin() + 1);

}

}

lex.push\_back(Lexeme("+", 0));

}

//лексический анализ

void lexic\_analis() {

//создаем таблицу состояний и функций, которые соответствую переходам между ними

std::vector<std::vector<std::pair<int, void (Parser::\*)(int a)>>> table(6);

for (int i = 0; i < table.size(); i++) {

table[i].resize(4);

for (int j = 0; j < 4; j++) {

table[i][j] = std::pair<int, void (Parser::\*)(int a)>(-1, 0);

}

}

table[0][0].first = 1; table[0][0].second = &Parser::t\_0\_0;

table[0][1].first = 2; table[0][1].second = &Parser::t\_0\_1;

table[0][2].first = 3; table[0][2].second = &Parser::t\_0\_2;

table[1][1].first = 2; table[1][1].second = &Parser::t\_1\_1;

table[1][2].first = 3; table[1][2].second = &Parser::t\_1\_2;

table[2][0].first = 1; table[2][0].second = &Parser::t\_2\_0;

table[2][2].first = 3; table[2][2].second = &Parser::t\_2\_2;

table[3][0].first = 1; table[3][0].second = &Parser::t\_3\_0;

table[3][2].first = 3; table[3][2].second = &Parser::t\_3\_2;

table[3][3].first = 4; table[3][3].second = &Parser::t\_3\_3;

table[4][1].first = 5; table[4][1].second = &Parser::t\_4\_1;

table[5][0].first = 1; table[5][0].second = &Parser::t\_5\_0;

table[5][2].first = 3; table[5][2].second = &Parser::t\_5\_2;

//сама работа конечного автомата

int status = 0;

for (int i = 0; i < lex.size(); i++) {

if (table[status][lex[i].getType()].first != -1) {

(this->\*table[status][lex[i].getType()].second)(i);

status = table[status][lex[i].getType()].first;

}

else {

std::cout << "in the lexeme number " << i << ": " << lex[i].str << " error";

throw std::invalid\_argument("in the lexeme number " + std::to\_string(i) + ": " + lex[i].str + " error");

}

}

}

public:

Parser(std::string input) {

input.erase(remove\_if(input.begin(), input.end(), isspace), input.end());

this->input = input;

syntaxis\_analis();

lexic\_analis();

}

Polinom getPolinom() {

return pol;

}

};

1. Структура данных (data structure) – это способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию **[1, c.30]**. [↑](#footnote-ref-1)