МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Иерархические списки

Студент гр. 8304 _____ Мешков М.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2019

Цель работы.

Получить опыт работы с иерархическими списками и их рекурсивной обработкой.

Постановка задачи.

1. Проанализировав условие задачи, разработать эффективный алгоритм для считывания данных и их обработки.

2. Сопоставить рекурсивное решение с итеративным решением задачи.

3. Сделать вывод о целесообразности и эффективности рекурсивного решения данной задачи.

Вариант 26.

Символьное дифференцирование алгебраического выражения, рассматриваемого как функция от одной из переменных. На входе выражение в иерархического списка и переменная, по которой следует дифференцировать. На выходе – производная исходного выражения.

Набор функций: +, -, *, ^, exp(). Упрощать не требуется.

Описание алгоритма.

Для решения поставленной задачи для начала нужно считать данные. Для этого были написаны функции (главная - метод load класса Expression), реализующие анализ выражения, на основе рекурсивного определения выражения (expr):

expr ::= binary_expr | unary_expr | (trivial_operand)

binary_expr ::= (expr_or_trivial_operand binary_oper expr_or_trivial_operand)

binary_oper ::= + | - | * | ^

unary_expr ::= (exp expr_or_trivial_operand)

expr_or_trivial_operand ::= expr | trivial_operand

trivial_operand ::= число | переменная

Эти функции переводят выражение в вид иерархического списка, который был реализован, используя созданный класс Node.

Также был написан набор функций (основная - метод derive в классе Expression) для выполнения обработки выражения, т.е. нахождения его производной. Аналогично функциям считывания, эти функции написаны с использованием рекурсии. Этот набор функций использует правила дифференцирования для выполнения своей задачи.

Для вывода обработанного выражения был написан метод toString в классе Expression.

Описание основных структур данных и функций.

Функция main осуществляет всё взаимодействие с пользователем, она приводит к созданию всех остальных используемых структур данных.

Класс Expression представляет собой класс для хранения и обработки выражения, внутри он хранит его в виде иерархического списка с помощью класса Node.

С помощью метода load класс Expression осуществляет "загрузку" выражения из строкового вида в вид удобный для работы (т.е. в вид иерархического списка). Этот метод делегирует почти всю работу приватным методам этого класса, которые легко найти по названию - они все начинаются со слова parse, эти методы переводят выражение из строкового представления используя рекурсивное определение выражения.

С помощью метода toString класс Expression осуществляет перевод выражения обратно в строковый вид.

Класс Expression также имеет метод derive для выполнения задания - дифференцирование выражения.

Класс Node используется для построения иерархического списка. Этот класс содержит свой тип, показывающий какое значение хранит экземпляр класса, одно из 4 значений (вид операции, число, имя переменной или вложенный список) и указатель на следующий элемент в текущем списке. Данный класс содержит функции для работы с отдельными экземплярами класса, а также он содержит функции для рекурсивной обработки

иерархических списков, которые выполняют основную работу по формированию строкового представления выражения и по дифференцированию выражения.

Тестирование.

Программа была протестирована. Ниже приведены основные проверочные входные данные (верхняя строка ввода - выражение, нижняя - переменная для дифференцирования).

Ввод	Вывод
((x * y) + (y ^ 2)) y	(((0 * y) + (x * 1)) + ((2 * (y ^ (2 - 1))) * 1))
((exp x)-9) x	(((exp x) * 1) - 0)
((exp 2) x	Error: Wrong expression.
(2 + 9)) x	Warning: Extra characters at the end of the expression. (0 + 0)
(x + (y + exp)) x	Error: Wrong expression.
(2e9 + (inf * x)) x	(0 + ((0 * x) + (inf * 1)))
(x ^ x) x	Error: Unsupported expression: Complex power.
(1.0009 + (2 ^ 3)) x	(0 + ((3 * (2 ^ (3 - 1))) * 0))
(x + (y - (x + z))) x	(1 + (0 - (1 + 0)))
4 + x x	Error: Wrong expression.
(x + y + z) y	Error: Wrong expression.

Выводы.

В ходе выполнения работы был получен опыт работы с иерархическими списками в контексте выполнения данной задачи. Был реализован рекурсивный

алгоритм ввода, обработки и вывода выражения. Итеративное решение данной задачи может быть гораздо сложнее как для реализации, так и для понимания.

ПРИЛОЖЕНИЕ А

Файл main.cpp

```
#include <iostream>
#include "expression.h"

int main()
{
   Expression expr;

   std::string exprStr;
   std::cout << "Enter expression: ";
   getline(std::cin, exprStr);

   std::string::size_type after;
   expr.load(exprStr, &after);
   if (!expr.isEmpty()) {
      if (after != exprStr.length())
            std::cerr << "Warning: Extra characters at the end of the expression." <<
std::endl;

      std::string variable;
      std::cout << "Enter target variable: ";
      std::cin >> variable;

      try {
         expr.derive(variable);
         std::cout << "Derived: " << expr.toString() << std::endl;
      } catch (Expression::ComplexPowerException &) {
         std::cerr << "Error: Unsupported expression: Complex power." << std::endl;
      }
```

```
    } else {
        std::cerr << "Error: Wrong expression." << std::endl;
    }

    return 0;
}
```

Файл expression.h

```
#pragma once

#include <string>
#include <memory>
#include <variant>
#include <exception>

class Expression {
public:
    /* "*after" will be changed to the next position after the expression. */
    void load(const std::string &exprStr, std::string::size_type *after = nullptr);

    bool isEmpty() const;

    std::string toString() const;

    class ComplexPowerException : public std::exception
    {};
    /* Differentiates by "variable".
     * May throw ComplexPowerException.
     */
    void derive(const std::string &variable);
```

```
private:
    enum class Operation {
        ADDITION,
        SUBTRACTION,
        MULTIPLICATION,
        POW,
        EXP,
    };

    class Node;
    std::shared_ptr<Node> m_head;

    /* "pos" will be changed to the next position after the chunk. */
    std::string readNextChunk(const std::string &exprStr, std::string::size_type &pos);

        /* The following several functions parse expressions using the recursive definitions:
     * expr ::= binary_expr | unary_expr | (trivial_operand)
     * binary_expr ::= (expr_or_trivial_operand binary_oper expr_or_trivial_operand)
     * binary_oper ::= + | - | * | ^
     * unary_expr ::= (exp expr_or_trivial_operand)
     * expr_or_trivial_operand ::= expr | trivial_operand
     * trivial_operand ::= number | variable
     */
    std::shared_ptr<Node> parseExpr(const std::string &exprStr, std::string::size_type &pos);
                std::shared_ptr<Node> parseBinaryExpr(const std::string &exprStr, std::string::size_type &pos);
                std::shared_ptr<Node> parseUnaryExpr(const std::string &exprStr, std::string::size_type &pos);
```

```
    std::shared_ptr<Node> parseExprOrTrivialOperand(const std::string &exprStr, std::string::size_type &pos);
        std::shared_ptr<Node> parseTrivialOperand(const std::string &exprStr, std::string::size_type &pos);
            std::shared_ptr<Node> parseNumber(const std::string &exprStr, std::string::size_type &pos);
            std::shared_ptr<Node> parseVariable(const std::string &exprStr, std::string::size_type &pos);
};

class Expression::Node {
public:
    Node() = default;
    Node(const Node &) = delete;
    Node &operator=(const Node &) = delete;

    static std::shared_ptr<Node> makeOperation(Operation operation);
    static std::shared_ptr<Node> makeNumber(double number);
    static std::shared_ptr<Node> makeVariable(std::string variable);
    static std::shared_ptr<Node> makeSubexpression(std::shared_ptr<Node> child);

    std::shared_ptr<Node> deepCopy(bool copySubsequent = false) const;

    void setOperation(Operation operation);
    void setNumber(double number);
    void setVariable(std::string variable);
    void setChild(std::shared_ptr<Node> child);

    enum class Type {
        OPERATION,
```

```
        NUMBER,
        VARIABLE,
        SUBEXPRESSION,
        NONE,
    };
    Node::Type getType() const;

    Operation getOperation() const;
    double getNumber() const;
    std::string getVariable() const;
    std::shared_ptr<Node> getChild() const;

    void setNext(std::shared_ptr<Node> next);
    std::shared_ptr<Node> getNext() const;

    std::string toString() const;
     bool containVariable(const std::string &variable, bool checkSubsequent = false)
const;
    void derive(const std::string &variable);

private:
    Node::Type m_type = Type::NONE;
    std::variant<Operation, double, std::string, std::shared_ptr<Node>> m_value;
    std::shared_ptr<Node> m_next = nullptr;

    void deriveSubexpression(const std::string &variable);
};
```

Файл expression.cpp

```
#include "expression.h"
#include <algorithm>
```

```
#include <sstream>

constexpr auto brokenExpressionMessage = "Broken expression.";

void Expression::load(const std::string &exprStr, std::string::size_type *after) {
    std::string::size_type pos = 0;
    m_head = parseExpr(exprStr, pos);
    if (after != nullptr)
        *after = pos;
}

bool Expression::isEmpty() const
{
    return m_head == nullptr;
}

std::string Expression::toString() const {
    if (m_head != nullptr)
        return m_head->toString();
    else
        return "";
}

void Expression::derive(const std::string &variable) {
    if (m_head != nullptr)
        m_head->derive(variable);
}

std::string Expression::readNextChunk(const std::string &exprStr,
std::string::size_type &pos) {
```

```
    std::string::size_type complexChunkLen = 0;
    for ( ; pos < exprStr.size(); pos++) {
        char c = exprStr[pos];
        if (isspace(c)) {
            if (complexChunkLen != 0)
                break;
            continue;
        }
        if (c == '(' || c == ')' || c == '+' || c == '-' || c == '*' || c == '^') {
            if (complexChunkLen != 0)
                break;
            return exprStr.substr(pos++, 1);
        }
        complexChunkLen++;
    }
    if (complexChunkLen != 0)
        return exprStr.substr(pos - complexChunkLen, complexChunkLen);
    else
        return {};
}

std::shared_ptr<Expression::Node> Expression::parseExpr(const std::string
&exprStr, std::string::size_type &pos)
{
    std::shared_ptr<Node> expr;

    expr = parseBinaryExpr(exprStr, pos);
    if (expr != nullptr)
        return expr;
```

```
    expr = parseUnaryExpr(exprStr, pos);
    if (expr != nullptr)
        return expr;

    auto oldPos = pos;
    auto nextChunk = readNextChunk(exprStr, pos);
    if (nextChunk == "(") {
        expr = parseTrivialOperand(exprStr, pos);
        auto nextChunk = readNextChunk(exprStr, pos);
        if (expr != nullptr && nextChunk == ")")
            return Node::makeSubexpression(expr);
    }
    pos = oldPos;
    return nullptr;
}

std::shared_ptr<Expression::Node> Expression::parseBinaryExpr(const std::string
&exprStr, std::string::size_type &pos)
{
    auto oldPos = pos;

    auto nextChunk = readNextChunk(exprStr, pos);
    if (nextChunk != "(") {
        pos = oldPos;
        return nullptr;
    }

    auto leftOperand = parseExprOrTrivialOperand(exprStr, pos);
    if (leftOperand == nullptr) {
        pos = oldPos;
```

```
    return nullptr;
}

nextChunk = readNextChunk(exprStr, pos);
auto operation = std::make_shared<Node>();
if (nextChunk == "+")
    operation->setOperation(Operation::ADDITION);
else if (nextChunk == "-")
    operation->setOperation(Operation::SUBTRACTION);
else if (nextChunk == "*")
    operation->setOperation(Operation::MULTIPLICATION);
else if (nextChunk == "^")
    operation->setOperation(Operation::POW);
else {
    pos = oldPos;
    return nullptr;
}

auto rightOperand = parseExprOrTrivialOperand(exprStr, pos);
if (rightOperand == nullptr) {
    pos = oldPos;
    return nullptr;
}

nextChunk = readNextChunk(exprStr, pos);
if (nextChunk != ")") {
    pos = oldPos;
    return nullptr;
}
```

```
    leftOperand->setNext(operation);
    operation->setNext(rightOperand);

    return Node::makeSubexpression(leftOperand);
}

std::shared_ptr<Expression::Node> Expression::parseUnaryExpr(const std::string
&exprStr, std::string::size_type &pos)
{
    auto oldPos = pos;

    auto nextChunk = readNextChunk(exprStr, pos);
    if (nextChunk != "(") {
        pos = oldPos;
        return nullptr;
    }

    nextChunk = readNextChunk(exprStr, pos);
    auto operation = std::make_shared<Node>();
    if (nextChunk == "exp") {
        operation->setOperation(Operation::EXP);
    } else {
        pos = oldPos;
        return nullptr;
    }

    auto operand = parseExprOrTrivialOperand(exprStr, pos);
    if (operand == nullptr) {
        pos = oldPos;
        return nullptr;
```

```
    }

    nextChunk = readNextChunk(exprStr, pos);
    if (nextChunk != ")") {
        pos = oldPos;
        return nullptr;
    }

    operation->setNext(operand);

    return Node::makeSubexpression(operation);
}

std::shared_ptr<Expression::Node>      Expression::parseExprOrTrivialOperand(const
std::string &exprStr, std::string::size_type &pos)
{
    std::shared_ptr<Node> node;

    node = parseExpr(exprStr, pos);
    if (node != nullptr)
        return node;

    node = parseTrivialOperand(exprStr, pos);
    if (node != nullptr)
        return node;

    return nullptr;
}
```

```
std::shared_ptr<Expression::Node> Expression::parseTrivialOperand(const
std::string &exprStr, std::string::size_type &pos)
{
    std::shared_ptr<Node> node;

    node = parseNumber(exprStr, pos);
    if (node != nullptr)
        return node;

    node = parseVariable(exprStr, pos);
    if (node != nullptr)
        return node;

    return nullptr;
}

std::shared_ptr<Expression::Node> Expression::parseNumber(const std::string
&exprStr, std::string::size_type &pos)
{
    auto oldPos = pos;

    auto nextChunk = readNextChunk(exprStr, pos);

    try {
        size_t after;
        auto number = std::stod(nextChunk, &after);
        if (after != nextChunk.length()) {
            pos = oldPos;
            return nullptr;
        }
```

```
        return Node::makeNumber(number);
    } catch (std::invalid_argument &) {
        pos = oldPos;
        return nullptr;
    } catch (std::out_of_range &) {
        return Node::makeNumber(std::numeric_limits<double>::infinity());
    }
}

std::shared_ptr<Expression::Node> Expression::parseVariable(const std::string &exprStr, std::string::size_type &pos)
{
    auto oldPos = pos;

    auto nextChunk = readNextChunk(exprStr, pos);

      if (std::all_of(nextChunk.begin(), nextChunk.end(), isalpha) && nextChunk != "exp")
        return Node::makeVariable(nextChunk);
    pos = oldPos;
    return nullptr;
}

std::shared_ptr<Expression::Node> Expression::Node::makeOperation(Expression::Operation operation)
{
    auto node = std::make_shared<Node>();
    node->setOperation(operation);
    return node;
}
```

```
std::shared_ptr<Expression::Node> Expression::Node::makeNumber(double number)
{
    auto node = std::make_shared<Node>();
    node->setNumber(number);
    return node;
}

std::shared_ptr<Expression::Node> Expression::Node::makeVariable(std::string
variable)
{
    auto node = std::make_shared<Node>();
    node->setVariable(variable);
    return node;
}

std::shared_ptr<Expression::Node>
Expression::Node::makeSubexpression(std::shared_ptr<Expression::Node> child)
{
    auto node = std::make_shared<Node>();
    node->setChild(child);
    return node;
}

std::shared_ptr<Expression::Node> Expression::Node::deepCopy(bool
copySubsequent) const {
    auto copy = std::make_shared<Node>();

    copy->m_type = m_type;
    if (m_type != Node::Type::SUBEXPRESSION)
```

```
        copy->m_value = m_value;
    else
        copy->m_value = getChild()->deepCopy(true);

    if (copySubsequent && m_next != nullptr)
        copy->m_next = m_next->deepCopy(true);
    else
        copy->m_next = nullptr;

    return copy;
}

void Expression::Node::setOperation(Expression::Operation operation)
{
    m_type = Node::Type::OPERATION;
    m_value = operation;
}

void Expression::Node::setNumber(double number)
{
    m_type = Node::Type::NUMBER;
    m_value = number;
}

void Expression::Node::setVariable(std::string variable)
{
    m_type = Node::Type::VARIABLE;
    m_value = variable;
}
```

```
void Expression::Node::setChild(std::shared_ptr<Expression::Node> child)
{
    if (child == nullptr)
        return;
    m_type = Node::Type::SUBEXPRESSION;
    m_value = child;
}

Expression::Node::Type Expression::Node::getType() const
{
    return m_type;
}

Expression::Operation Expression::Node::getOperation() const {
    return std::get<Operation>(m_value);
}

double Expression::Node::getNumber() const {
    return std::get<double>(m_value);
}

std::string Expression::Node::getVariable() const {
    return std::get<std::string>(m_value);
}

std::shared_ptr<Expression::Node> Expression::Node::getChild() const {
    return std::get<std::shared_ptr<Node>>(m_value);
}

void Expression::Node::setNext(std::shared_ptr<Expression::Node> next)
```

```
{
    m_next = next;
}

std::shared_ptr<Expression::Node> Expression::Node::getNext() const
{
    return m_next;
}

std::string Expression::Node::toString() const
{
    auto callToString = [](std::shared_ptr<Node> node) -> std::string {
        if (node != nullptr)
            return node->toString();
        else
            return "";
    };

    switch (getType()) {
    case Node::Type::SUBEXPRESSION:
        return "(" + callToString(getChild()) + ")" + callToString(getNext());
    case Node::Type::OPERATION:
        switch (getOperation()) {
        case Operation::ADDITION:
            return " + " + callToString(getNext());
        case Operation::SUBTRACTION:
            return " - " + callToString(getNext());
        case Operation::MULTIPLICATION:
            return " * " + callToString(getNext());
        case Operation::POW:
```

```
            return " ^ " + callToString(getNext());
        case Operation::EXP:
            return "exp " + callToString(getNext());
        }
        return ""; // Never will be reached
    case Node::Type::NUMBER: {
        std::ostringstream ss;
        ss << getNumber();
        return ss.str() + callToString(getNext());
    }
    case Node::Type::VARIABLE:
        return getVariable() + callToString(getNext());
    case Node::Type::NONE:
        return "" + callToString(getNext());
    }
    return ""; // Never will be reached
}

bool Expression::Node::containVariable(const std::string &variable, bool
checkSubsequent) const
{
    auto callContainVariable = [&](std::shared_ptr<Node> node) -> bool {
        if (node != nullptr)
            return node->containVariable(variable, true);
        else
            return false;
    };

    if (getType() == Node::Type::VARIABLE && getVariable() == variable)
        return true;
```

```
                    if (getType() == Node::Type::SUBEXPRESSION &&
callContainVariable(getChild()))
        return true;
    if (checkSubsequent && callContainVariable(getNext()))
        return true;
    return false;
}

void Expression::Node::derive(const std::string &variable)
{
    switch (getType()) {
    case Node::Type::NUMBER:
        /* Formula: da/dx = 0 */
        setNumber(0.0);
        return;
    case Node::Type::VARIABLE:
        /* Formula: dx/dx = 1 */
        /* Formula: dy/dx = 0 */
        if (getVariable() == variable)
            setNumber(1.0);
        else
            setNumber(0.0);
        return;
    case Node::Type::SUBEXPRESSION: {
        deriveSubexpression(variable);
        return;
    }
    default:
        return;
    }
```

```
}

void Expression::Node::deriveSubexpression(const std::string &variable)
{
    auto callDerive = [&variable](std::shared_ptr<Node> node) {
        if (node != nullptr)
            node->derive(variable);
    };

    auto first = getChild();
    auto second = first->getNext();
    if (second == nullptr) {
        callDerive(first);
        return;
    }
    if (first->getType() == Node::Type::OPERATION) {
        auto operation = first->getOperation();
        switch (operation) {
        case Operation::EXP: {
            /* Formula: (exp(a))' = ((exp(a)) * (a)') */
            auto derivedA = second->deepCopy();
            callDerive(derivedA);
                                                    auto multiplication =
Node::makeOperation(Operation::MULTIPLICATION);
            multiplication->setNext(derivedA);
            auto derivedExp = Node::makeSubexpression(first);
            derivedExp->setNext(multiplication);
            setChild(derivedExp);
            return;
        }
```

```
    case Operation::ADDITION:
    case Operation::SUBTRACTION:
    case Operation::MULTIPLICATION:
    case Operation::POW:
        throw std::logic_error(brokenExpressionMessage);
    }
    return;
} else if (second->getType() == Node::Type::OPERATION) {
    auto third = second->getNext();
    if (third == nullptr)
        throw std::logic_error(brokenExpressionMessage);
    auto operation = second->getOperation();
    switch (operation) {
    case Operation::ADDITION:
    case Operation::SUBTRACTION:
        /* Formula: (a + b)' = ((a)' + (b)') */
        /* Formula: (a - b)' = ((a)' - (b)') */
        callDerive(first);
        callDerive(third);
        return;
    case Operation::MULTIPLICATION: {
        /* Formula: (a * b)' = (((a)' * b) + (a * (b)')) */
        auto leftAddend = deepCopy();
        callDerive(leftAddend->getChild());
        callDerive(third);
        auto rightAddend = Node::makeSubexpression(first);
        auto addition = Node::makeOperation(Operation::ADDITION);
        addition->setNext(rightAddend);
        leftAddend->setNext(addition);
        setChild(leftAddend);
```

```
            return;
        }
        case Operation::POW: {
            if (third->containVariable(variable))
                throw ComplexPowerException();

            /* Formula: (a ^ b)' = ((b * (a ^ (b - 1))) * (a)') */
            auto derivedA = first->deepCopy();
            callDerive(derivedA);
            auto b = third->deepCopy();

            auto aPowBMinusOne = Node::makeSubexpression(first);
            setChild(nullptr);
            auto subtraction = Node::makeOperation(Operation::SUBTRACTION);
            auto one = Node::makeNumber(1.0);
            subtraction->setNext(one);
            third->setNext(subtraction);
            second->setNext(Node::makeSubexpression(third));

                                                    auto leftMultiplication =
Node::makeOperation(Operation::MULTIPLICATION);
            leftMultiplication->setNext(aPowBMinusOne);
            b->setNext(leftMultiplication);
            auto bMulAPowBMinusOne = Node::makeSubexpression(b);

                                                    auto rightMultiplication =
Node::makeOperation(Operation::MULTIPLICATION);
            rightMultiplication->setNext(derivedA);
            bMulAPowBMinusOne->setNext(rightMultiplication);
```

```
            setChild(bMulAPowBMinusOne);
            return;
        }
        case Operation::EXP:
            throw std::logic_error(brokenExpressionMessage);
        }
        return;
    }
    throw std::logic_error(brokenExpressionMessage);
}
```