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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и струтуры данных»

Тема: Вычисления в различных формах записи

Студент гр. 1301 _____ Устинов Г.А.

Преподаватель _____ Родионова Е.А.

Санкт-Петербург

2022

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Устинов Г.А.

Группа 1301

Тема работы: Вычисления в различных формаъ записи

Исходные данные: Реализовать вычисление арифметических выражений с бинарными операциями {+,-,*,/,^} и числовыми операндами, записанных в префиксной/инфиксной/постфиксной записи.

Содержание пояснительной записки:

Содержание

Введение

Описание реализуемых классов и функций

Пример работы

Заключение

Список использованных источников

Предполагаемый объем пояснительной записки:

Не менее 10 страниц.

Дата выдачи задания: 14.11.2022

Дата сдачи реферата: 26.12.2022

Дата защиты реферата: 26.12.2022

Студент _____ Устинов Г.А.

Преподаватель _____ Родионова Е.А.

АННОТАЦИЯ

В данной курсовой работе исследуются методы считывания и расчёта арифметический выражений, записанных в рзличной форме (инфиксной, префиксной, постфиксной) и способ абстрактного представления в виде синтаксического дерева. Также реализуется считывание выражений в заданных формах, их вычисление и предоставление результатов в интерактивной форме

SUMMARY

Present course work is dedicated to research of methods parsing and calculation of arithmetical expressions written in different forms (infix, prefix and postfix) and means of abstract representation in form of syntax tree. Also a program being developed that implements parsing of arithmetical expressions, their evaluation and presentation of results in an interactive form.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе требуется еализовать вычисление арифметических выражений с бинарными операциями {+,-,*,/,^} и числовыми операндами, записанных в префиксной/инфиксной/постфиксной записи. Для выполнения поставленной задачи будет использовано представление выражения в виде дерева, где листья — конкретные значения, а узлы — действия, котрые требуется совершить.

1. ОПИСАНИЕ РЕАЛИЗУЕМЫХ КЛАССОВ И ФУНКЦИЙ

Для выполнения работы были разработаны несколько классов, которые представляют различные сущности и имеют различные ответсвенности в выполняемой задаче.

1.1. Арифметические выражения и представление их структуры

Структура выражений сведена к двум понятиям — оператор и операнд. Операнд представляется иерархией интерфейса *Operand*, оператор — *Operator.*

Интерфейс *Operand* представляют классы *Constant* — объект, значение которого известно на этапе формирования выражения, — и *Expression* — объект, значение которого не тривиально и требует проведения вычислений для получения результатов. Объекты *Exression* хранят оператор — наименее приоритетный оператор, представляющий последнюю выполняемую операцию перед получением результата, для вычисления которого может понадобится предварительное вычисление других выражений, которые этот оператор хранит как операнды.

Интейрфейс *Operator* реализуется как *UnaryOperator* и *BinaryOperator* — объекты унарного и бинарного оператора соответсвенно. Они различаются не только количеством аргументов, но и логикой считывания из строковой записи.

Комбинация этих двух иерархий позволяет представить выражение в универсальной форме двоичного дерева — в котором чем ниже наъодится вершина, тем позднее при вычислении выражения будет произведено это действие. При этом листья дерева — это исключительно константы, заданные в числовом виде, или заранее, при компиляции. При вычислении выражения будет произведён поиск в глубину — таким образом будут соблюдены приоритеты операций.

1.2. Считывание арифметического выражения из текста

Для сопоставления строковой последовательности некоторому оператору или числовому значению необходима таблица, пары строка — выражение. Для этого был реализован статический класс *ParsingTable*. Таблица представляет

собой три списка — список бинарных операторов, список унарных операторов и список констант. Все три списка заполняются перед началом работы.

Также список предоставляет функции, позволяющие производить проверку, соответствует ли строке оператор или константа, а также получать объекты, соответствующие переданной строке.

На базе работы с этим классом предоставляются три группы функций, которые осуществляют считавание трёх типов выражений — записанных в инфиксной, префиксной и постфиксной форме. Для каждой группы существует одна входная функция, и вспомогательные, соответсвующие парадигме динамического программирования, и позволяющие переиспользовать одну и ту же логику для разных выражений. Ключевое отличие составляет только группа чтения постфиксной формы записи — для удобства работы, строка «разворачивается», и не смотря на формально ту же логику работы в таком случае, что и считывание префиксной формы, необходимо учитывать обратный порядок символов.

При чтении префиксной и постфиксной форм записи дерево выражений строится на месте — первым предполагается наименее приоритетный двоичный оператор, что позволяет тут же начать строить дерево выражения, рекурсивно передавая остаток строки для построения выражения.

В случае инфиксной формы записи, установить приоритетность операций возможно только после полного обращения строки в объекты, с которыми можно работать — поэтому считаваются два списка — список операторов и список операндов, причём список операндов всегда на еденицу длиннее списка операторов. После этого, в списке операторов находится оператор с наименьшим приритетом — он будет корнем дерева — списки разбиваются относительно этого элемента, и для каждого подсписка алгоритм запускается снова. После завершения работы получается дерево, обход в глубину которого и даст значение выражения.

1.3. Основной цикл работы

В основном цикле программа находится в одном из трёх состояний чтения — по форме записи выражения, которое она будет обрабатывать. Текущий режим отображается в приглашении ко вводу. Режим изменяется командой формата *:<mode>*, где *mode* — режим работы. В каждом из режимов программа ожидает ввод в режиме интерактивной командной строки. Получая на вход выражение, программа анализирует его, и выдаёт число — результат вычисления выражения. Получив на вход пустую строку, программа завершается.

Временная сложность

Чтение каждой формы записи имеет свою временную сложность.

Для инфиксной формы, если принять ввод верным, то есть есть входное выражение не содержит ошибок, то процесс считывания выражения является линейным, пусть и с весьма большими постоянными расходами. Каждый символ исходной строки будет обработан не более двух раз — при нахождении подстроки, в котрой содержится очередной элемент, и при копирования во временный буфер для получения объектов от таблицы. Такая простота достигается благодаря принципу чередования операторов и операндов в инфиксной форме записи, что позволяет сразу начинать считывание нужного типа.

В префиксной и постфикной формах записи после каждого оператора или операнда может следовать выражение произвольной глубины, что не позволяет делать преждевременных выводов о следующей записи. При попытке поиска осмысленной подстроки, каждый символ может быть считан столько раз, сколько перед ним записано операторов и операндов, что даёт в примерном приближении квадратичное время выполнения.

2. ПРИМЕР РАБОТЫ

В приведённом ниже примере на вход программе подаётся одно и то же выражение, записаное во всех трёх формах записи. Для смены режима последующей формы записи программе даётся команда для смена режима. В конце, получив пустой ввод, программа завершается.

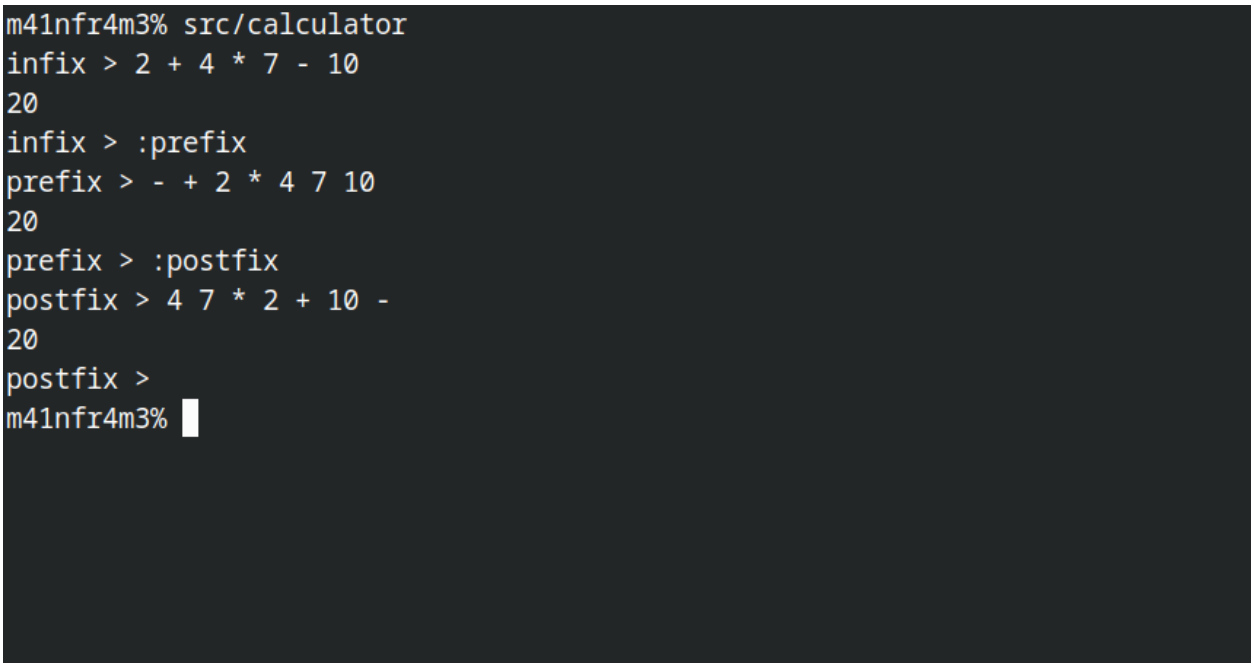

Рисунок 1 — Пример работы программы

ЗАКЛЮЧЕНИЕ

В данной работе были изучены отличия и особенности каждой из форм записи. Инфиксная форма показала себя как имеющая наименьшую временную сложность, не смотря на то, что на первый взгляд она казалась наиболее сложной для считывания, в отличие от префиксной и инфиксной форм записи.

Также было установленно, что древовидная структура представления выражения не является лучшей для представления префиксных и постфиксных выражений. В таких случаях лучше подойдёт такая структура, как стек.

Однако, древовидное представление позволяет приводить все выражения к одному и тому же виду, ввиду чего упрощается обобщёный анализ выражений.

Была написана программа, реализующая считывание и расчёт выражений, а также предлагающая пользователю три режима работы на выбор.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Веб-версия книги Problem Solving with Algorithms and Data Structures Using Python // Перевод книги Problem Solving with Algorithms and Data Structures Using Python. URL: http://aliev.me/runestone/BasicDS/InfixPrefixandPostfixExpressions.html (дата обращения 26.12.2022).
2. Свободная энциклопедия, поддерживаемая сообществом // Википедия — свободная энциклопедия. URL: https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C (дата обращения 25.12.2022).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл **src/main.cpp**

```
#include <iomanip>
#include <iostream>
#include <memory>
#include <string>

#include "calculation-tree.hh"
#include "parsing.hh"
#include "parsing-exceptions.hh"

using namespace calculation;
using namespace parsing;
using namespace std;

enum Mode { kInfix, kPrefix, kPostfix };

int main()
{
   init_table();
   cout << std::defaultfloat;
   string buffer;
   shared_ptr<Operand> res;
   Mode mode = kInfix;
   do {
       switch (mode) {
       case kInfix:
           cout << "infix ";
           break;
       case kPrefix:
           cout << "prefix ";
           break;
       case kPostfix:
           cout << "postfix ";
           break;
       }
       cout << "> ";
       getline(cin, buffer);
       if (buffer.empty())
           break;
       if (buffer[0] == ':') {
           if (buffer.substr(1) == "infix")
               mode = kInfix;
           else if (buffer.substr(1) == "prefix")
               mode = kPrefix;
```

```
            else if (buffer.substr(1) == "postfix")
                mode = kPostfix;
            else
                cout << "Unknown mode \"" << buffer.substr(1) << '"' <<
endl;
        } else {
            try {
                switch (mode) {
                case kInfix:
                    res = parse_infix_expression(buffer);
                    break;
                case kPrefix:
                    res = parse_prefix_expression(buffer);
                    break;
                case kPostfix:
                    res = parse_postfix_expression(buffer);
                    break;
                }
            } catch (const ParserError &e) {
                cout << "Invalid expression." << endl;
                continue;
            }
            cout << res->evaluate() << endl;
        }
    } while (true);
    return 0;
}
```

Файл **src/parsing.hh**

```
#pragma once
#ifndef PARSING_HH
#define PARSING_HH

#include <string>

#include "calculation-tree.hh"

namespace parsing {

void init_table();

/*
* Returns an expression, evaluation of which will calculate the
expression
* stored in the string.
*/
std::shared_ptr<calculation::Operand> parse_infix_expression(const
```

```
std::string &str, size_t start = 0);

std::shared_ptr<calculation::Operand> parse_prefix_expression(const
std::string &str);

std::shared_ptr<calculation::Operand> parse_postfix_expression(const
std::string &str);

}   // namespace parsing

#endif  // PARSING_HH
```

Файл **src/parsing.cpp**

```
#include "parsing.hh"

#include <cmath>
#include <functional>
#include <list>
#include <memory>
#include <string>

#include "calculation-tree.hh"
#include "parsing-exceptions.hh"
#include "parsing-table.hh"

namespace parsing {

using table = ParsingTable;

using namespace std;

using calculation::Operand;
using calculation::Constant;
using calculation::Expression;

using calculation::Operator;
using calculation::UnaryOperator;
using calculation::BinaryOperator;

void init_table()
{
   table::register_constant("pi", 3.141592653589793);
   table::register_constant("e",  2.718281828459045);

   table::register_unary("-", negate<double>());

   table::register_binary("^", (double (*)(double, double))pow, 0);
```

```
    table::register_binary("*", multiplies<double>(), 1);
    table::register_binary("/", divides<double>(), 1);
    table::register_binary("+", plus<double>(), 2);
    table::register_binary("-", minus<double>(), 2);
}


size_t skip_spaces(const string &str, size_t start)
{
    const size_t len = str.size();
    size_t pos = start;
    while (pos < len && isspace(str[pos]))
        ++pos;
    return pos;
}

shared_ptr<Constant> parse_value(const string &str, size_t &start)
{
    size_t processed = 0;
    double val;
    try {
        string buffer(str, start, str.size() - start);
        val = stod(buffer, &processed);
    } catch (out_of_range &) {
        throw TooBigNumber(start);
    }
    shared_ptr<Constant> res = make_shared<Constant>(
        val,
        string(str, start, processed)
    );
    start += processed;
    return res;
}

shared_ptr<Constant> parse_constant(const string &str, size_t &start)
{
    const size_t len = str.length();
    size_t pos = start;
    string copy;
    copy += str[pos];
    while (!table::is_constant(copy)) {
        if (++pos == len)
            throw UnexpectedEndOfExpression(pos);
        copy += str[pos];
    }
    start = pos + 1;
    return table::get_constant(copy);
}
```

```
shared_ptr<UnaryOperator> parse_unary(const string &str, size_t &start)
{
   const size_t len = str.length();
   size_t pos = skip_spaces(str, start);
   string copy;
   copy += str[pos];
   while (!table::is_unary_operator(copy)) {
       if (++pos == len)
           throw UnexpectedEndOfExpression(pos);
       copy += str[pos];
   }
   start = pos + 1;
   return table::get_unary_operator(copy);
}

shared_ptr<Operand> parse_infix_operand(const string &str, size_t &start)

{
   const size_t len = str.length();
   if (start >= len)
       throw UnexpectedEndOfExpression(start);
   size_t pos = skip_spaces(str, start);
   if (pos == len)
       throw OperandExpectationUnsatisfied(pos);

   size_t backup_pos = pos;
   shared_ptr<Operand> res;
   if (str[pos] == '(') {
       size_t counter = 1;
       for (++pos; pos < len && counter > 0; ++pos) {
           if (str[pos] == '(')
               ++counter;
           else if (str[pos] == ')')
               --counter;
       }
       if (pos == len && counter != 0)
           throw UnexpectedEndOfExpression(pos);
       string inner(str, backup_pos + 1, pos - backup_pos - 2);
       try {
           res = parse_infix_expression(inner);
       } catch (ParserError &e) {
           // Restoring absolute position in the string and re-throwing
           e.position += backup_pos + 1;
           throw e;
       }
   } else if (table::is_starting_digit(str[pos])) {
       res = parse_value(str, pos);
```

```
    } else {
        try {
            shared_ptr<UnaryOperator> tmp = parse_unary(str, pos);
            tmp->set_operand(parse_infix_operand(str, pos));

            shared_ptr<Expression> exp(new Expression);
            exp->set_root(tmp);
            res = exp;
        } catch (const ParserError&) {
            pos = backup_pos;
            try {
                res = parse_constant(str, pos);
            } catch (const UnexpectedEndOfExpression&) {
                throw OperandExpectationUnsatisfied(backup_pos);
            }
        }
    }
    start = pos;
    return res;
}


shared_ptr<BinaryOperator> parse_operator(const string &str, size_t
&start)
{
    const size_t len = str.length();
    if (start >= len)
        throw UnexpectedEndOfExpression(len);
    size_t pos = skip_spaces(str, start);
    if (pos == len)
        throw BinaryExpectationUnsatisfied(start);

    string copy;
    copy += str[pos];
    while (!table::is_binary_operator(copy)) {
        if (++pos == len)
            throw UnexpectedEndOfExpression(pos);
        copy += str[pos];
    }
    start = pos + 1;
    return table::get_binary_operator(copy);
}

shared_ptr<Operand>
create_tree_from_parser_lists(list<shared_ptr<Operand>> operands,
list<shared_ptr<BinaryOpera
tor>> operators)
{
```

```
    if (operands.size() == 1)
        return *operands.begin();
    shared_ptr<Expression> res = make_shared<Expression>();
    list<shared_ptr<Operand>>::iterator operand = operands.begin();
    list<shared_ptr<Operand>>::iterator left_operand = operand++;
    list<shared_ptr<BinaryOperator>>::iterator op_it = operators.begin();
    list<shared_ptr<BinaryOperator>>::iterator hang_point = op_it++;
    while (op_it != operators.end()) {
        if ((*hang_point)->order() <= (*op_it)->order()) {
            hang_point = op_it;
            left_operand = operand;
        }
        ++op_it;
        ++operand;
    }
    res->set_root(*hang_point);
    (*hang_point)->set_left(create_tree_from_parser_lists(
        {operands.begin(), ++left_operand},
        {operators.begin(), hang_point}));
    (*hang_point)->set_right(create_tree_from_parser_lists(
        {left_operand, operands.end()},
        {++hang_point, operators.end()}));
    return res;
}

shared_ptr<Operand> parse_infix_expression(const string &str, size_t
start)
{
    const size_t len = str.length();
    if (len == 0)
        return shared_ptr<Operand>(new Constant(0));
    /*
     * At first, there will definetly be either a constant or an unary
operator
     * with its own operand. If not, there's no parsable expression
anyway,
     * so exception fallback from parse_infix_operand is acceptable.
     */
    size_t pos = start;
    shared_ptr<Operand> tmpoperand;
    shared_ptr<BinaryOperator> tmpoperator;
    list<shared_ptr<BinaryOperator>> operators;
    list<shared_ptr<Operand>> operands;
    do {
        pos = skip_spaces(str, pos);
        tmpoperand = parse_infix_operand(str, pos);
        operands.push_back(tmpoperand);
        pos = skip_spaces(str, pos);
```

```
        if (pos < len) {
            tmpoperator = parse_operator(str, pos);
            operators.push_back(tmpoperator);
        } else {
            break;
        }
    } while (true);

    shared_ptr<Operand> res;
    if (operands.size() == 1)
        res = *operands.begin();
    else
        res = create_tree_from_parser_lists(operands, operators);
    return res;
}

shared_ptr<Operand> parse_prefix_operand(const string &str, size_t
&start)
{
    const size_t len = str.length();
    if (start >= len)
        throw UnexpectedEndOfExpression(start);
    size_t pos = skip_spaces(str, start);
    if (pos == len)
        throw OperandExpectationUnsatisfied(pos);

    size_t backup_pos = pos;
    shared_ptr<Operand> res;
    if (table::is_starting_digit(str[pos])) {
        res = parse_value(str, pos);
    } else {
        try {
            res = parse_constant(str, pos);
        } catch (const UnexpectedEndOfExpression&) {
            throw OperandExpectationUnsatisfied(backup_pos);
        }
    }
    start = pos;
    return res;
}

shared_ptr<Operand> parse_prefix_expression_worker(const string &str,
size_t &start)
{
    const size_t len = str.length();
    size_t pos = skip_spaces(str, start);
    if (len - pos == 0)
        throw OperandExpectationUnsatisfied(pos);
```

```
    shared_ptr<Operand> res;
    shared_ptr<Operator> tmpoperator;
    size_t backup_pos = pos;
    try {
        shared_ptr<BinaryOperator> tmp = parse_operator(str, pos);
        tmp->set_left(parse_prefix_expression_worker(str, pos));
        tmp->set_right(parse_prefix_expression_worker(str, pos));
        shared_ptr<Expression> exp = make_shared<Expression>();
        exp->set_root(tmp);
        res = exp;
    } catch (const ParserError &) {
        pos = backup_pos;
        try {
            shared_ptr<UnaryOperator> tmp = parse_unary(str, pos);
            tmp->set_operand(parse_prefix_expression_worker(str, pos));
            shared_ptr<Expression> exp = make_shared<Expression>();
            exp->set_root(tmp);
            res = exp;
        } catch (const ParserError &) {
            pos = backup_pos;
            res = parse_prefix_operand(str, pos);
        }
    }
    start = pos;
    return res;
}

shared_ptr<Operand> parse_prefix_expression(const string &str)
{
    size_t start = 0;
    return parse_prefix_expression_worker(str, start);
}


shared_ptr<UnaryOperator> parse_postfix_unary(const string &rev, size_t
&start)
{
    const size_t len = rev.length();
    size_t pos = skip_spaces(rev, start);
    string copy;
    copy += rev[pos];
    while (!table::is_unary_operator(copy)) {
        if (++pos >= len)
            throw UnexpectedEndOfExpression(pos);
        copy.insert(copy.begin(), rev[pos]);
    }
    start = pos + 1;
    return table::get_unary_operator(copy);
```

```
}

shared_ptr<BinaryOperator> parse_postfix_operator(const string &rev,
size_t &start)
{
   const size_t len = rev.length();
   if (start >= len)
       throw UnexpectedEndOfExpression(len);
   size_t pos = skip_spaces(rev, start);
   if (pos >= len)
       throw BinaryExpectationUnsatisfied(start);

   string copy;
   copy += rev[pos];
   while (!table::is_binary_operator(copy)) {
       if (++pos >= len)
           throw UnexpectedEndOfExpression(pos);
       copy.insert(copy.begin(), rev[pos]);
   }
   start = pos + 1;
   return table::get_binary_operator(copy);
}

shared_ptr<Constant> parse_postfix_value(const string &rev, size_t
&start)
{
   const size_t len = rev.length();
   size_t processed = 0;
   double val;
   try {
       string buffer;
       char c;
       while (start + processed < len) {
           c = rev[start + processed];
           if (!isdigit(c) && c != '.')
               break;
           buffer.insert(buffer.begin(), c);
           ++processed;
       }
       val = stod(buffer, &processed);
   } catch (out_of_range &) {
       throw TooBigNumber(start);
   }
   shared_ptr<Constant> res = make_shared<Constant>(
       val,
       string(rev, start - processed + 1, processed)
   );
   start += processed;
```

```
   return res;
}

shared_ptr<Constant> parse_postfix_constant(const string &rev, size_t
&start)
{
   const size_t len = rev.length();
   size_t pos = start;
   string copy;
   copy += rev[pos];
   while (!table::is_constant(copy)) {
       if (++pos >= len)
           throw UnexpectedEndOfExpression(pos);
       copy.insert(copy.begin(), rev[pos]);
   }
   start = pos + 1;
   return table::get_constant(copy);
}

shared_ptr<Operand> parse_postfix_operand(const string &str, size_t
&start)
{
   const size_t len = str.length();
   if (start >= len)
       throw UnexpectedEndOfExpression(start);
   size_t pos = skip_spaces(str, start);
   if (pos >= len)
       throw OperandExpectationUnsatisfied(pos);

   size_t backup_pos = pos;
   shared_ptr<Operand> res;
   try {
       res = parse_postfix_constant(str, pos);
   } catch (const ParserError &) {
       pos = backup_pos;
       try {
           res = parse_postfix_value(str, pos);
       } catch (const UnexpectedEndOfExpression&) {
           throw OperandExpectationUnsatisfied(backup_pos);
       }
   }
   start = pos;
   return res;
}

std::shared_ptr<calculation::Operand>
parse_postfix_expression_worker(const std::string &rev, size_t &start)
{
```

```
    const size_t len = rev.length();
    size_t pos = skip_spaces(rev, start);
    if (len - pos == 0)
        throw OperandExpectationUnsatisfied(start);
    shared_ptr<Operand> res;
    shared_ptr<Operator> tmpoperator;
    size_t backup_pos = pos;
    try {
        shared_ptr<BinaryOperator> tmp = parse_postfix_operator(rev, pos);

        tmp->set_right(parse_postfix_expression_worker(rev, pos));
        tmp->set_left(parse_postfix_expression_worker(rev, pos));
        shared_ptr<Expression> exp = make_shared<Expression>();
        exp->set_root(tmp);
        res = exp;
    } catch (const ParserError &) {
        pos = backup_pos;
        try {
            shared_ptr<UnaryOperator> tmp = parse_postfix_unary(rev, pos);

            tmp->set_operand(parse_postfix_expression_worker(rev, pos));
            shared_ptr<Expression> exp = make_shared<Expression>();
            exp->set_root(tmp);
            res = exp;
        } catch (const ParserError &) {
            pos = backup_pos;
            res = parse_postfix_operand(rev, pos);
        }
    }
    start = pos;
    return res;
}

std::shared_ptr<calculation::Operand> parse_postfix_expression(const
std::string &str)
{
    size_t start = 0;
    string rev(str.rbegin(), str.rend());
    return parse_postfix_expression_worker(rev, start);
}

}   // namespace parsing
```

Файл **src/parsing-table.hh**

```
#pragma once
#ifndef PARSING_TABLE_HH
#define PARSING_TABLE_HH
```

```
#include <cctype>
#include <functional>
#include <list>
#include <memory>
#include <stdexcept>
#include <string>

#include "calculation-tree.hh"

namespace parsing {

/*
* A static table that holds entries on how to decode text into math
*/
class ParsingTable {
public:
   class InvalidNameError;
   class NameSearchError;

   ParsingTable() = delete;
   ParsingTable(const ParsingTable &) = delete;
   ParsingTable(ParsingTable &&) = delete;

   static bool is_valid_name(const std::string &name);
   static bool is_starting_digit(char c) { return std::isdigit(c); }
   static bool is_digit(char c) { return std::isdigit(c) || c == '.'; }

   static void register_constant(const std::string &name, const double
value);
   static void register_unary(const std::string &name,
std::function<double (double)> f);
   static void register_binary(const std::string &name,
std::function<double (double, double)> f, unsigned order
);

   static bool is_constant(const std::string &name);
   static bool is_unary_operator(const std::string &name);
   static bool is_binary_operator(const std::string &name);

   static std::shared_ptr<calculation::Constant> get_constant(const
std::string &name);
   static std::shared_ptr<calculation::UnaryOperator>
get_unary_operator(const std::string &name);
   static std::shared_ptr<calculation::BinaryOperator>
get_binary_operator(const std::string &name);
private:
   struct ConstantEntry;
```

```
    struct UnaryOperatorEntry;
    struct BinaryOperatorEntry;

    ~ParsingTable() = default;

    static std::list<ConstantEntry> constants_;
    static std::list<UnaryOperatorEntry> unary_operators_;
    static std::list<BinaryOperatorEntry> binary_operators_;
};


class ParsingTable::InvalidNameError : public std::invalid_argument {
public:
    InvalidNameError(const std::string &name)
        : invalid_argument("Invalid name \"" + name + "\".")
    {}
};


class ParsingTable::NameSearchError : public std::invalid_argument {
public:
    NameSearchError(const std::string &name)
        : invalid_argument("No such name found \"" + name + "\".")
    {}
};


struct ParsingTable::ConstantEntry {
    ConstantEntry() = delete;
    ConstantEntry(const std::string &name, const double value)
        : data(value, name)
    {}

    const calculation::Constant data;
};


struct ParsingTable::UnaryOperatorEntry {
    UnaryOperatorEntry() = delete;
    UnaryOperatorEntry(const std::string &name,
std::function<double(double)> f)
        : data(name, f)
    {}

    const calculation::UnaryOperator data;
};
```

```
struct ParsingTable::BinaryOperatorEntry {
   BinaryOperatorEntry() = delete;
   BinaryOperatorEntry(const std::string &name,
std::function<double(double, double)> f, unsigned order)
       : data(name, f, order)
   {}

   const calculation::BinaryOperator data;
};

}   // namespace parsing

#endif  // PARSING_TABLE_HH
```

Файл **src/parsing-table.cpp**

```
#include "parsing-table.hh"

#include <cctype>
#include <stdexcept>

#include "parsing-table.hh"

using namespace std;

namespace parsing {

list<ParsingTable::ConstantEntry> ParsingTable::constants_;
list<ParsingTable::UnaryOperatorEntry> ParsingTable::unary_operators_;
list<ParsingTable::BinaryOperatorEntry> ParsingTable::binary_operators_;


bool ParsingTable::is_valid_name(const string &name)
{
   if (name.length() == 0)
       return false;
   if (isdigit(name[0]))
       return false;
   for (auto c : name) {
       if (!isgraph(c))
           return false;
   }
   return true;
}


void ParsingTable::register_constant(const string &name, const double
value)
```

```
{
   if (!is_valid_name(name))
       throw InvalidNameError(name);
   constants_.push_back({name, value});
}

void ParsingTable::register_unary(const string &name, function<double
(double)> f)
{
   if (!is_valid_name(name))
       throw InvalidNameError(name);
   if (!f)
       throw invalid_argument("Operator cannot be null.");
   unary_operators_.push_back({name, f});
}

void ParsingTable::register_binary(const string &name, function<double
(double, double)> f, unsigned order)
{
   if (!is_valid_name(name))
       throw InvalidNameError(name);
   if (!f)
       throw invalid_argument("Operator cannot be null.");
   binary_operators_.push_back({name, f, order});
}


bool ParsingTable::is_constant(const string &name)
{
   if (!is_valid_name(name))
       return false;
   for (auto &c : constants_) {
       if (c.data.str() == name)
           return true;
   }
   return false;
}

bool ParsingTable::is_unary_operator(const string &name)
{
   if (!is_valid_name(name))
       return false;
   for (auto &c : unary_operators_) {
       if (c.data.repr() == name)
           return true;
   }
   return false;
}
```

```
bool ParsingTable::is_binary_operator(const string &name)
{
   if (!is_valid_name(name))
       return false;
   for (auto &c : binary_operators_) {
       if (c.data.repr() == name)
           return true;
   }
   return false;
}


shared_ptr<calculation::Constant> ParsingTable::get_constant(const string
&name)
{
   if (!is_valid_name(name))
       throw InvalidNameError(name);
   for (auto &c : constants_) {
       if (name == c.data.str())
           return shared_ptr<calculation::Constant>(new
calculation::Constant(c.data));
   }
   throw NameSearchError(name);
}

shared_ptr<calculation::UnaryOperator>
ParsingTable::get_unary_operator(const string &name)
{
   if (!is_valid_name(name))
       throw InvalidNameError(name);
   for (auto &op : unary_operators_) {
       if (name == op.data.repr())
           return shared_ptr<calculation::UnaryOperator>(new
calculation::UnaryOperator(op.data));
   }
   throw NameSearchError(name);
}

shared_ptr<calculation::BinaryOperator>
ParsingTable::get_binary_operator(const string &name)
{
   if (!is_valid_name(name))
       throw InvalidNameError(name);
   for (auto &op : binary_operators_) {
       if (name == op.data.repr())
           return shared_ptr<calculation::BinaryOperator>(new
calculation::BinaryOperator(op.data));
```

```
    }
    throw NameSearchError(name);
}

}   // namespace parsing
```

Файл **src/parsing-exceptions.hh**

```
#pragma once
#ifndef PARSING_EXCEPTIONS_HH
#define PARSING_EXCEPTIONS_HH

#include <stdexcept>

namespace parsing {

class ParserError : public std::runtime_error {
public:
    ParserError(const std::string about, size_t pos)
        : runtime_error(about), position(pos)
    {}

    size_t position;
};

class SyntaxError : public ParserError {
public:
    SyntaxError(const std::string about, size_t pos)
        : ParserError(about, pos)
    {}
};

class UnexpectedEndOfExpression : public SyntaxError {
public:
    UnexpectedEndOfExpression(size_t pos)
        : SyntaxError("Unexpected end of string expression.", pos)
    {}
    UnexpectedEndOfExpression(const std::string about, size_t pos)
        : SyntaxError(about, pos)
    {}
};

class UnsatisfiedExpectation : public SyntaxError {
public:
    UnsatisfiedExpectation(const std::string about, size_t pos)
        : SyntaxError(about, pos)
    {}
};
```

```
class OperandExpectationUnsatisfied : public UnsatisfiedExpectation {
public:
   OperandExpectationUnsatisfied(size_t pos)
       : UnsatisfiedExpectation("Operand was expected, but couldn't be
found.", pos)
   {}
};

class BinaryExpectationUnsatisfied : public UnsatisfiedExpectation {
public:
   BinaryExpectationUnsatisfied(size_t pos)
       : UnsatisfiedExpectation("Binary operator was expected, but
couldn't be found.", pos)
   {}
};

class TooBigNumber : public ParserError {
public:
   TooBigNumber(size_t pos)
       : ParserError("Given constant is too big to be stored.", pos)
   {}
};

}   // namespace parsing

#endif  // PARSING_EXCEPTIONS_HH
```

Файл **src/calculation-tree.hh**

```
#pragma once
#ifndef CALCULATION_TREE_HH
#define CALCULATION_TREE_HH

#include <functional>
#include <memory>
#include <string>

namespace calculation {


/*
* Operands are something that operators work with. Operand can be
* evaluated to get it's value.
*/
class Operand {
public:
   virtual ~Operand() = default;
```

```
   virtual double evaluate() const = 0;

   virtual std::string str() const = 0;
};


/*
* Operator takes its operands and passes them to an underlying math
function.
*/
class Operator {
public:
   virtual ~Operator() = default;

   virtual double calculate() const = 0;

   virtual std::string str() const = 0;
   virtual std::string repr() const = 0;
};


/*
* Constant is an operand, which value is known at parsetime. Constant's
* value cannot be changed.
*/
class Constant : public Operand {
public:
   Constant() = delete;
   Constant(double value) : value_(value) {}
   Constant(double value, const std::string &name)
       : name_(name), value_(value)
   {}
   Constant(const Constant &other)
       : name_(other.name_), value_(other.value_)
   {}
   Constant(Constant &&other) : name_(other.name_), value_(other.value_)
{}

   double evaluate() const { return value_; }

   std::string str() const;
private:
   std::string name_;
   double value_;
};
```

```
/*
* Expression is an operand that needs to calculate a few operators
* itself, before it can tell its value.
*/
class Expression : public Operand {
public:
   Expression() = default;
   /*
    * Sets a root operator for the expression calculation tree.
    */
   void set_root(const std::shared_ptr<Operator> &op) { root_ = op; }
   std::shared_ptr<Operator> get_root() { return root_; }

   double evaluate() const;

   std::string str() const { return root_->str(); }
private:
   std::shared_ptr<Operator> root_;
};


class UnaryOperator : public Operator {
public:

   UnaryOperator() = delete;
   // UnaryOperator(double (*f)(double)) : operator_(f) {}
   UnaryOperator(const std::string &str, std::function<double(double)> f)

       : operator_(f), str_(str)
   {}
   UnaryOperator(const UnaryOperator &other)
       : operator_(other.operator_), str_(other.str_)
   {}
   UnaryOperator(UnaryOperator &&other)
       : operator_(other.operator_), str_(other.str_)
   {
       operand_.swap(other.operand_);
   }

   void set_operand(std::shared_ptr<Operand> op) { operand_ = op; }
   std::shared_ptr<Operand> get_operand() { return operand_; }

   double calculate() const;

   std::string repr() const { return str_; }
   std::string str() const { return str_ + " " + operand_->str(); }
private:
   std::function<double(double)> operator_;
```

```
    std::string str_;
    std::shared_ptr<Operand> operand_;
};


class BinaryOperator : public Operator {
public:
    BinaryOperator() = delete;
    BinaryOperator(const std::string &str, std::function<double(double,
double)> f, unsigned order)
        : operator_(f), str_(str), order_(order)
    {}
    BinaryOperator(const BinaryOperator &other)
        : operator_(other.operator_), str_(other.str_),
order_(other.order_)
    {}
    BinaryOperator(BinaryOperator &&other)
        : operator_(other.operator_), str_(other.str_),
order_(other.order_)
    {
        left_.swap(other.left_);
        right_.swap(other.right_);
    }

    void set_left(std::shared_ptr<Operand> op) { left_ = op; }
    std::shared_ptr<Operand> get_left() { return left_; }

    void set_right(std::shared_ptr<Operand> op) { right_ = op; }
    std::shared_ptr<Operand> get_right() { return right_; }

    unsigned order() const { return order_; }

    std::string repr() const { return str_; }
    std::string str() const { return str_ + " " + left_->str() + " " +
right_->str(); }

    double calculate() const;
private:
    std::function<double(double, double)> operator_;
    std::string str_;

    std::shared_ptr<Operand> left_;
    std::shared_ptr<Operand> right_;

    unsigned order_;
};
```

```
}   // namespace calculation

#endif  // CALCULATION_TREE_HH
```

Файл **src/calculation-tree.cpp**

```
#include "calculation-tree.hh"

#include <stdexcept>
#include <string>

namespace calculation {

double Expression::evaluate() const
{
   if (!root_)
       throw std::logic_error("Evaluating empty expression.");
   return root_->calculate();
}


std::string Constant::str() const
{
   if (name_.empty())
       return std::to_string(value_);
   return name_;
}


double UnaryOperator::calculate() const
{
   if (!operator_)
       throw std::logic_error("Calculation of non-bind operator.");
   else if (!operand_)
       throw std::logic_error("Calculation of operator with no
operand.");
   return operator_(operand_->evaluate());
}


double BinaryOperator::calculate() const
{
   if (!operator_)
       throw std::logic_error("Calculation of non-bind operator.");
   else if (!left_)
       throw std::logic_error("Calculation of operator with no left
operand.");
   else if (!right_)
```

```
       throw std::logic_error("Calculation of operator with no right
operand.");
   return operator_(left_->evaluate(), right_->evaluate());
}
}   // namespace calculation
```

Файл **src/CMakeLists.txt**

```
add_library(calculation-tree STATIC)
target_sources(calculation-tree
       PRIVATE calculation-tree.cpp
       PUBLIC calculation-tree.hh
)

add_library(parsing-table STATIC)
target_sources(parsing-table
       PRIVATE parsing-table.cpp
       PUBLIC parsing-table.hh
)

add_library(parsing STATIC)
target_sources(parsing
       PRIVATE parsing.cpp
       PUBLIC parsing.hh
)

add_executable(calculator)
target_sources(calculator
       PRIVATE main.cpp
)

target_link_libraries(calculator parsing parsing-table calculation-tree)
```

Файл **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.16)

project(ads-cw-2022 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
add_compile_options(
       -Wall
       -Wextra
       -Wpedantic
)
add_subdirectory(src)
```