МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Иерархические списки

Студент гр. 9304 _____ Арутюнян В.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с понятием иерархического списка, реализовать его, решить поставленную задачу на его основе.

Постановка задачи.

Вариант 22.

Пусть алгебраическое выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме (<*операция*> <*аргументы*>). Аргументов может быть до двух включительно.

Доступные операции: *+, -, *, sqrt(), log(,).*

Требуется выполнить проверку синтаксической корректности введенного выражения. И выполнить простую проверку *log(,)*, если это возможно.

Выполнение работы.

Программа на вход ожидает строку для анализа сразу после флага “-s”, иначе ожидается путь до файла со строкой для обработки. Возможно совместное использование:

./lab2 some/path/1 -s “some_string_1” -s “some_string_2” some/path/2

При таком вызове программа обработает строку из файла *some/path/1*, затем строку *some_string_1*, после строку *some_string_2*, далее строку из файла *some/path/2*.

Класс AlgebraicExpression.

Данный класс был реализован для обработки и анализа входной строки. В нём хранятся следующие поля:

1. *expr_* – анализируем строка;
2. *err_* – ошибка, которая могла возникнуть в ходе работы анализатора, изначально устанавливается в состояние “без ошибок”;

Для начала анализа теста, достаточно создать экземпляр класса, который будет содержать входную строку, и вызвать метод *Analyze()*. В данном методе происходит создание объекта класса *HierarchicalList*, вызов метода *CheckCorrect()* и обработка возможных ошибок анализа строки. Последнее происходит с помощью еще одного реализованного класса *MyException()*.

Внутри *CheckCorrect()* происходит проверка уже созданного объекта класса *HierarchicalList* на синтаксическую корректность. В основе проверки лежит то, что выражение из иерархического списка должно представляться одним из следующий способов:

1. Число/Переменная/Иерархический список (что-то одно);
2. Операция (*+, -, *, log*), затем выражение из п. 1, после выражение из п. 1;
3. Операция (*sqrt*), затем выражение из п. 1.

Обнаруженный иерархический список внутри другого списка проверяется рекурсивно вызовом метода *CheckCorrect()*. Если указанные требования нарушаются, то генерируется исключение.

Также в данном классе существует метод GetError(), возвращающий ошибку типа MyException.

Класс Node.

Данный класс является реализацией элемента иерархического списка. В нём хранятся следующие поля:

1. *next_* – умный указатель на следующий элемент списка;
2. *object_* – *std::variant*, который может содержать число, или тип объекта (переменная или тип операции), или умный указатель на иерархический список.

Также есть конструкторы для каждого элемента, который может хранится в *std::variant* объекта *object_*. В классе есть методы, необходимые для проверки синтаксической корректности введенной строки:

1. *GetPtrHierarchicalList()* – получение константного иерархического списка;

2. *GetNext()* – получение следующего элемента списка;
3. *GetOperation()* – получение операции, хранящейся в *object_*;
4. *GetNumber()* – получение числа, хранящего в *object_*;
5. *IsHierarchicalList()* – установление того, является ли элемент, который хранится в *object_* иерархическим списком;
6. *IsOperation()* – установление того, является ли элемент, который хранится в *object_* операцией;
7. *IsVariable()* – установление того, является ли элемент, который хранится в *object_* переменной;
8. *IsNumber()* – установление того, является ли элемент, который хранится в *object_* числом.

Класс HierarchicalList.

Данный класс является реализацией иерархического списка. В нём хранятся следующие поля:

1. *names_* – статическая переменная, хранящая имена логических переменных в виде строк;
2. *head_* – умный указатель на первый элемент иерархического списка;
3. *end_* – умный указатель на последний элемент иерархического списка.

В классе есть конструктор по умолчанию и конструктор для *std::string_view*. В последнем происходит преобразование входной строки в иерархический список. Для того чтобы получить следующий непробельный символ используется метод класса *GetFirstNonSpace()*. Также в классе существуют следующие методы:

1. *AppendArgumentOfLogOrSqrt()* – добавление в конец переданного иерархического списка одного из аргументов логарифма или единственный аргумент операции *sqrt()*;
2. *Append()* – шаблонный метод добавления элемента в конец списка;
3. *GetDistanceToClosingBracket()* – получение расстояния от открывающей скобки до соответствующей закрывающей скобки;

4. *GetDistanceToComma()* – получение расстояние от открывающей скобки до запятой в операции *log()*;
5. *CompareStrings()* – сравнение строки со строками из *names_*;
6. *IsVariable()* – установление того, является ли начало переданной строки переменной;
7. *IsNumber()* – установление того, является ли начало переданной строки числом;
8. *IsSqrt()* – установление того, является ли начало переданной строки префиксом операции *sqrt()* («sqrt»);
9. *IsLog()* – установление того, является ли начало переданной строки префиксом операции *log()* («log»);
10. *GetHead()* – получение константного первого элемента иерархического списка.

Обработка входной строки происходит следующим образом: последовательно обрабатываются символы, в зависимости от текущего символа выполняются различные действия:

1. При получение некоторых символов операции (+, -, *) вызывается метод *Append()*, который добавляет операцию в иерархический список;
2. При встрече открывающей скобки создается новый иерархический список для строки, начинающейся от встреченной открывающей скобки до закрывающей скобки, найденной с помощью метода *GetDistanceToClosingBracket()*, затем вызывается метод *Append()*, добавляющий созданный список в исходный;
3. При встрече префикса операции *log()* или *sqrt()*, создается новый иерархический список, в который добавляется префикс операции, а затем последовательно аргументы логарифма или единственный аргумент *sqrt()*. Далее созданный список добавляется в исходный;
4. Также если предыдущие пункты не были обработаны, то последовательно проверяется строка на наличие числа в ней и наличие переменной, если число не было найдено;

5. В ином случае считается, что встречен некорректный символ. Генерируется исключение.

Также проверяются скобки, при их недостаточном количестве генерируется исключение.

Элемент *end_* необходим для более простой и эффективной реализации метода *Append()*.

enum class NamesType необходим для удобного и понятного обращения к элементам массива names_.

Класс MyException.

Данный класс необходим для хранения кода сгенерированной ошибки и дополнительной строки с пояснениями. Здесь же определен метод получения кода ошибки – *GetCode()* и перегружен оператор вывода.

enum class ErrorCode необходим для понятной записи кода ошибки.

Программа выводит “TRUE”, если входная строка являлась корректным алгебраическим выражением, иначе выводит “FALSE” в первой строке и тип ошибки – во второй. Исходный код находится в приложении А.

Далее будет проиллюстрировано хранение алгебраического выражения в иерархическом списке.

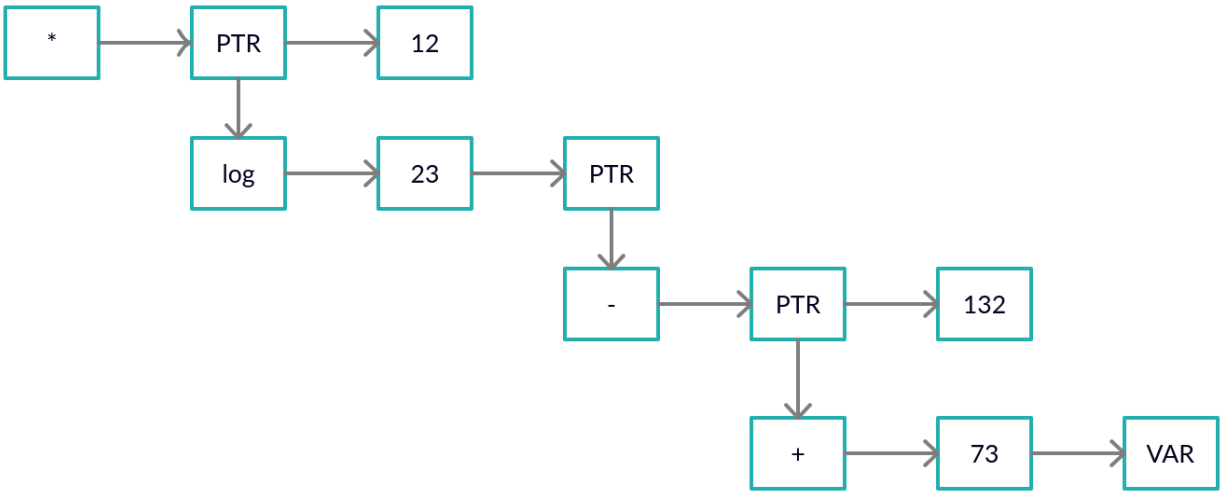

Рисунок 1 – Иллюстрация хранения алгебраического выражения в иерархическом списке

На Рисунке 1 представлено следующее алгебраическое выражение: "* log(23, – (+ 73 var) 132) 12".

Тестирование.

Программу можно собрать через *Makefile* командой *make*, после этого создается исполняемый файл *lab2*. Существует несколько вариантов провести тестирование:

1. Вызвать *lab2*, указав путь до файла с тестом, либо передать флаг "*-s*", а затем строку, которую требуется проанализировать, в кавычках;
2. Запустить python-скрипт – *testing.py*, в котором можно изменять параметры для тестирования, например, количество тестов, их расположение, расположение эталонных ответов, расположение ответов, полученных от программы.

Далее будет представлена таблица тестирования с несколькими тестами.

Таблица 1. Примеры входных и выходных данных

№	Входные данные	Выходные данные
1	(+ 0 (*1 (-a b)))	TRUE
2	* log(1, * (+1 alse) 132) 12	FALSE ValueError: invalid argument of the logarithm
3	log((1 + 0), 12)	FALSE SyntaxError: invalid syntax
4	log(2 10)	FALSE SyntaxError: expected comma not found
5	+ var123	FALSE SyntaxError: not enough arguments
6	+ 123var	FALSE SyntaxError: invalid syntax
7	* sqrt(log(+3-12, sqrt(- log(2, 102) (+ 12 var991_129))))	FALSE SyntaxError: not enough arguments
8	(2183ijo/1)	FALSE SyntaxError: invalid syntax
9	log(2, 0)	FALSE ValueError: invalid argument of the logarithm
10	* sqrt(log(+ 3-12, sqrt(- log(2, 102) (+ 12 var991_129)))) _____VAr12_1	TRUE

Выводы.

В ходе выполнения лабораторной работы был реализован иерархический список и решена поставленная задача на его основе. Была разработана программа, анализирующая алгебраическое выражение. Использование иерархического списка, с одной стороны, оправдывается более простой обработкой элементов, но, с другой стороны, требует больше памяти при создании объекта, а также при рекурсивном анализе.

ПРИЛОЖЕНИЕ А

main.cpp

```
#include <fstream>
#include <iostream>
#include <string>

#include "../lib/algebraic_expression.h"
#include "../lib/my_exception.h"

void PrintAnalyzeResult(bool is_correct_expr, const MyException&
err) {
  if (is_correct_expr) {
    std::cout << "TRUE\n";
  } else {
    std::cout << "FALSE\n" << err << '\n';
  }
}

int main(int argc, char** argv) {
  if (argc == 1) {
    std::cout << "Too small arguments.\n"
              << "A algebraic expression is expected after \"-s\"
flag, "
              << "otherwise a file path is expected.\n\n"
              << "example: ./lab2 -s \"+ 0 (* 1 2)\"
Tests/test/test1.txt\n";
  } else {
    bool is_string = false;

    for (int i = 1; i < argc; ++i) {
      std::string arg = argv[i];

      if (is_string) {
        is_string = false;
        AlgebraicExpression expr(arg);
        bool is_correct_expr = expr.Analyze();
        PrintAnalyzeResult(is_correct_expr, expr.GetError());

      } else if (arg == "-s") {
        is_string = true;

      } else {
        std::ifstream file_in(arg);
```

```
        if (file_in.is_open()) {
          std::string str;
          std::getline(file_in, str);
          AlgebraicExpression expr(str);
          bool is_correct_expr = expr.Analyze();
          PrintAnalyzeResult(is_correct_expr, expr.GetError());
          file_in.close();

        } else {
          std::cout << "Couldn't open the file.\n";
        }
      }
    }
  }
}
```

algebraic_expression.h

```
#ifndef ALGEBRAIC_EXPRESSION_H_
#define ALGEBRAIC_EXPRESSION_H_

#include <string>

#include "my_exception.h"
#include "hierarchical_list.h"

class AlgebraicExpression {
 public:
  AlgebraicExpression(std::string_view expr);
  MyException GetError();
  bool Analyze();
  bool CheckCorrect(const HierarchicalList& object);
  ~AlgebraicExpression() = default;

 private:
  std::string expr_;
  MyException err_;
};

#endif  // ALGEBRAIC_EXPRESSION_H_
```

algebraic_expression.cpp

```
#include "../lib/algebraic_expression.h"

AlgebraicExpression::AlgebraicExpression(std::string_view expr)
```

```
    : expr_(expr), err_(ErrorCode::kNone) {}

MyException AlgebraicExpression::GetError() { return err_; }

bool AlgebraicExpression::Analyze() {
  try {
    HierarchicalList list_expr(expr_);
    CheckCorrect(list_expr);
  } catch (const MyException& err) {
    err_ = err;
    return false;
  }
  return true;
}

bool AlgebraicExpression::CheckCorrect(const HierarchicalList&
object) {
  Node curr = object.GetHead();

  // one argument - Hierarchical List
  if (curr.IsHierarchicalList()) {
    CheckCorrect(curr.GetHierarchicalList());

    // one operation - sqrt
  } else if (curr.IsOperation() && curr.GetOperation() ==
TypeOfObject::kSqrt) {
    if (curr.GetNext() == nullptr || curr.GetNext()-
>IsOperation()) {
      throw MyException(ErrorCode::kSyntaxError,
                        "not enough arguments or invalid syntax");
    } else {
      // next item is a hierarchical list
      if (curr.GetNext()->IsHierarchicalList()) {
        CheckCorrect(curr.GetNext()->GetHierarchicalList());
      }
      // shift by 1 elements
      curr = *(curr.GetNext());
    }
    // any operation
  } else if (curr.IsOperation()) {
    if (curr.GetNext() == nullptr || curr.GetNext()->GetNext() ==
nullptr) {
      throw MyException(ErrorCode::kSyntaxError, "not enough
arguments");
    }
```

```
    Node next = *(curr.GetNext());
    Node next_x2 = *(next.GetNext());

    if (next.IsOperation() || next_x2.IsOperation()) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
argument");
    }

    // log(number_1, number_2)
    if (curr.GetOperation() == TypeOfObject::kLog &&
        ((next.IsNumber() &&
          (next.GetNumber() <= 0 || next.GetNumber() == 1)) ||
         (next_x2.IsNumber() && next_x2.GetNumber() <= 0))) {
      throw MyException(ErrorCode::kValueError,
                        "invalid argument of the logarithm");
    } else {
      // next item is a hierarchical list
      if (next.IsHierarchicalList()) {
        CheckCorrect(next.GetHierarchicalList());
      }
      // next_x2 item is a hierarchical list
      if (next_x2.IsHierarchicalList()) {
        CheckCorrect(next_x2.GetHierarchicalList());
      }
      // shift by 2 elements
      curr = next_x2;
    }
  }

  if (curr.GetNext() != nullptr) {
    throw MyException(ErrorCode::kSyntaxError, "invalid syntax");
  }

  return true;
}
```

hierarchical_list.h

```
#ifndef HIERARCHICAL_LIST_H_
#define HIERARCHICAL_LIST_H_

#include <algorithm>
#include <iostream>
#include <string>

#include "my_exception.h"
```

```
#include "node.h"

enum class NamesType { kLog = 0, kSqrt };

class HierarchicalList {
 public:
  HierarchicalList();
  HierarchicalList(std::string_view str);

  const Node& GetHead() const;

  ~HierarchicalList() = default;

 private:
  static const std::string names_[2];
  std::shared_ptr<Node> head_;
  std::shared_ptr<Node> end_;

  void AppendArgumentOfLogOrSqrt(std::string_view str, int begin,
int end,
                                 char sym,

std::shared_ptr<HierarchicalList>& h_list);

  template <typename T>
  void Append(T object);

  int GetDistanceToClosingBracket(std::string_view str);
  int GetFirstNonSpace(std::string_view str, int pos);
  int GetDistanceToComma(std::string_view str);

  bool CompareStrings(std::string_view str, NamesType type);
  int IsVariable(std::string_view str);
  int IsNumber(std::string_view str);
  bool IsSqrt(std::string_view str);
  bool IsLog(std::string_view str);
};

#endif  // HIERARCHICAL_LIST_H_
```

hierarchical_list.cpp

```
#include "../lib/hierarchical_list.h"

const std::string HierarchicalList::names_[] = {"log", "sqrt"};
```

```
HierarchicalList::HierarchicalList() : head_(nullptr),
end_(nullptr) {}

HierarchicalList::HierarchicalList(std::string_view str)
    : head_(nullptr), end_(nullptr) {
  int curr_index = GetFirstNonSpace(str, 0);
  bool first_iteration = true;
  int count_of_brackets = 0;
  int shift = 1;

  if (curr_index == str.size()) {
    throw MyException(ErrorCode::kSyntaxError, "empty string");
  }

  while (curr_index != str.size()) {
    unsigned char curr_sym = str[curr_index];

    if (curr_sym == '(' && first_iteration) {
      ++count_of_brackets;

    } else if (curr_sym == '(') {
      int distance_to_closing_bracket =
          GetDistanceToClosingBracket(str.substr(curr_index,
str.size()));
      std::string_view temp =
          str.substr(curr_index, 1 + distance_to_closing_bracket);
      std::shared_ptr<HierarchicalList> h_list(new
HierarchicalList(temp));
      Append(h_list);
      shift = 1 + distance_to_closing_bracket;

    } else if (curr_sym == ')') {
      --count_of_brackets;

    } else if (curr_sym == '+') {
      Append(TypeOfObject::kPlus);

    } else if (curr_sym == '-' && !(curr_index + 1 != str.size()
&&
                                    isdigit(str[curr_index + 1])))
{
      Append(TypeOfObject::kMinus);

    } else if (curr_sym == '*') {
      Append(TypeOfObject::kMultiply);
```

```
    } else if (IsLog(str.substr(curr_index, str.size()))) {
      std::shared_ptr<HierarchicalList> h_list(new
HierarchicalList());
      h_list->Append(TypeOfObject::kLog);  // adding the operation

      curr_index = GetFirstNonSpace(str, curr_index + 3);  //
shift to "("

      if (curr_index == str.size() || str[curr_index] != '(') {
        throw MyException(ErrorCode::kSyntaxError,
                          "expected opening bracket not found");
      }

      std::string_view log_str = str.substr(curr_index,
str.size());
      int distance_to_closing_bracket =
GetDistanceToClosingBracket(log_str);
      int distance_to_comma = GetDistanceToComma(log_str);

      // adding the first argument
      AppendArgumentOfLogOrSqrt(str, curr_index,
distance_to_comma, ')',
                                h_list);
      // adding the second argument
      AppendArgumentOfLogOrSqrt(str, curr_index +
distance_to_comma,
                                distance_to_closing_bracket -
distance_to_comma,
                                '(', h_list);
      Append(h_list);
      shift = distance_to_closing_bracket + 1;

    } else if (IsSqrt(str.substr(curr_index, str.size()))) {
      std::shared_ptr<HierarchicalList> h_list(new
HierarchicalList());
      h_list->Append(TypeOfObject::kSqrt);  // adding the
operation

      curr_index = GetFirstNonSpace(str, curr_index + 4);  //
shift to "("

      if (curr_index == str.size() || str[curr_index] != '(') {
        throw MyException(ErrorCode::kSyntaxError,
                          "expected opening bracket not found");
      }
```

```
      std::string_view sqrt_str = str.substr(curr_index,
str.size());
      int distance_to_closing_bracket =
GetDistanceToClosingBracket(sqrt_str);

      // adding the argument
      AppendArgumentOfLogOrSqrt(str, curr_index,
distance_to_closing_bracket,
                                ' ', h_list);
      Append(h_list);
      shift = distance_to_closing_bracket + 1;

    } else if (shift = IsNumber(str.substr(curr_index,
str.size()))) {
      Append(stoll((std::string)str.substr(curr_index,
str.size())));

    } else if (shift = IsVariable(str.substr(curr_index,
str.size()))) {
      Append(TypeOfObject::kVariable);

    } else {
      throw MyException(ErrorCode::kSyntaxError, "invalid
syntax");
    }

    if (count_of_brackets < 0) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
syntax");
    }

    first_iteration = false;
    curr_index += shift;
    shift = 1;  // default shift

    curr_index = GetFirstNonSpace(str, curr_index);
  }

  if (count_of_brackets > 0) {
    throw MyException(ErrorCode::kSyntaxError, "invalid syntax");
  }
}

const Node& HierarchicalList::GetHead() const {
  if (head_ == nullptr) {
    throw MyException(ErrorCode::kRuntimeError, "empty list");
```

```
  }
  return *head_;
}

void HierarchicalList::AppendArgumentOfLogOrSqrt(
    std::string_view str, int begin, int dist, char sym,
    std::shared_ptr<HierarchicalList>& h_list) {
  std::string temp(str.substr(begin, 1 + dist));
  std::string_view temp_view = temp.substr(1, dist - 1);
  if (!temp_view.size()) {
    throw MyException(ErrorCode::kSyntaxError, "not enough
arguments");
  }
  temp_view.remove_prefix(GetFirstNonSpace(temp_view, 0));
  int shift = IsNumber(temp_view);
  if (GetFirstNonSpace(temp_view, shift) ==
      temp_view.size()) {  // adding a number
    h_list->Append(stoll((std::string)temp_view));
  } else {
    if (sym == '(') {
      temp.front() = sym;
    } else if (sym == ')') {
      temp.back() = sym;
    }
    std::shared_ptr<HierarchicalList> h_list_child(new
HierarchicalList(temp));
    h_list->Append(h_list_child);
  }
}

template <typename T>
void HierarchicalList::Append(T object) {
  if (head_ == nullptr) {
    head_ = std::make_shared<Node>(object);
    end_ = head_;
  } else {
    end_->next_ = std::make_shared<Node>(object);
    end_ = end_->next_;
  }
}

int HierarchicalList::GetDistanceToClosingBracket(std::string_view
str) {
  int count_of_brackets = 0;
  bool first_bracket = true;
  int closing_bracket = str.size();
```

```
  for (int i = 0; i < str.size(); ++i) {
    if (str[i] == '(') {
      ++count_of_brackets;
      first_bracket = false;
    } else if (str[i] == ')') {
      --count_of_brackets;
    }

    if (count_of_brackets < 0) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
syntax");
    }

    if (!count_of_brackets && !first_bracket) {
      closing_bracket = i;
      break;
    }
  }

  if (count_of_brackets > 0) {
    throw MyException(ErrorCode::kSyntaxError, "invalid syntax");
  }

  return closing_bracket;
}

int HierarchicalList::GetFirstNonSpace(std::string_view str, int
pos) {
  for (; pos < str.size(); ++pos) {
    if (!isspace(str[pos])) return pos;
  }
  return str.size();
}

int HierarchicalList::GetDistanceToComma(std::string_view str) {
  int count_of_brackets = 0;
  bool first_bracket = true;
  int distance_to_comma = str.size();

  for (int i = 0; i < str.size(); ++i) {
    if (str[i] == '(') {
      ++count_of_brackets;
      first_bracket = false;
    } else if (str[i] == ')') {
      --count_of_brackets;
```

```
    }

    if (count_of_brackets < 0) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
syntax");
    }

    if (count_of_brackets == 1 && str[i] == ',') {
      distance_to_comma = i;
      break;
    }
  }

  if (distance_to_comma == str.size()) {
    throw MyException(ErrorCode::kSyntaxError, "expected comma not
found");
  }

  return distance_to_comma;
}

bool HierarchicalList::CompareStrings(std::string_view str,
NamesType type) {
  return str.substr(0, names_[(int)type].size()) ==
names_[(int)type];
}

int HierarchicalList::IsVariable(std::string_view str) {
  int i = 0;
  for (; i < str.size() &&
         (std::isalpha(str[i]) || (str[i] == '_') ||
(isdigit(str[i]) && i));
       ++i)
    ;
  return i;
}

int HierarchicalList::IsNumber(std::string_view str) {
  bool is_negative = false;
  int i = 0;

  if (1 < str.size() && str[i] == '-' && (i + 1) < str.size() &&
      isdigit(str[i + 1])) {
    is_negative = true;
    ++i;
  }
```

```
  for (; i < str.size() && isdigit(str[i]); ++i)
    ;

  if ((i < str.size() && (str[i] == '_' || std::isalpha(str[i]))))
{
    i = 0;
  }

  if (i && ((!is_negative && i > 18) || (is_negative && i > 19)))
{
    throw MyException(ErrorCode::kValueError, "too big number");
  }

  return i;
}

bool HierarchicalList::IsSqrt(std::string_view str) {
  return CompareStrings(str, NamesType::kSqrt);
}

bool HierarchicalList::IsLog(std::string_view str) {
  return CompareStrings(str, NamesType::kLog);
}
```

node.h

```
#ifndef NODE_H_
#define NODE_H_

#include <iostream>
#include <memory>
#include <variant>

#include "my_exception.h"

enum class TypeOfObject {
  kPlus = 0,
  kMinus,
  kMultiply,
  kSqrt,
  kLog,
  kVariable
};

class HierarchicalList;
```

```
class Node {
 public:
  Node(std::shared_ptr<HierarchicalList>& object);
  Node(const TypeOfObject& object);
  Node(const long long& object);

  const HierarchicalList& GetHierarchicalList();
  const std::shared_ptr<Node>& GetNext();
  const TypeOfObject& GetOperation();
  int GetNumber();

  bool IsHierarchicalList();
  bool IsOperation();
  bool IsVariable();
  bool IsNumber();

  ~Node() = default;

 private:
  friend class HierarchicalList;
  std::shared_ptr<Node> next_;
  std::variant<std::shared_ptr<HierarchicalList>, TypeOfObject,
long long>
      object_;
};
#endif  // NODE_H_
```

node.cpp

```
#include "../lib/node.h"

Node::Node(std::shared_ptr<HierarchicalList>& object)
    : next_(nullptr), object_(object) {}
Node::Node(const TypeOfObject& object) : next_(nullptr),
object_(object) {}
Node::Node(const long long& object) : next_(nullptr),
object_(object) {}

const HierarchicalList& Node::GetHierarchicalList() {
  if (!IsHierarchicalList()) {
    throw MyException(ErrorCode::kRuntimeError,
                      "getting a different type of object");
  }
  return *(std::get<std::shared_ptr<HierarchicalList>>(object_));
}
```

```
const std::shared_ptr<Node>& Node::GetNext() {
  return next_;
}

const TypeOfObject& Node::GetOperation() {
  if (!IsOperation()) {
    throw MyException(ErrorCode::kRuntimeError,
                      "getting a different type of object");
  }
  return std::get<TypeOfObject>(object_);
}

int Node::GetNumber() {
  if (!IsNumber()) {
    throw MyException(ErrorCode::kRuntimeError,
                      "getting a different type of object");
  }
  return std::get<long long>(object_);
}

bool Node::IsHierarchicalList() {
  return
std::holds_alternative<std::shared_ptr<HierarchicalList>>(object_)
;
}

bool Node::IsOperation() {
  return std::holds_alternative<TypeOfObject>(object_) &&
         std::get<TypeOfObject>(object_) !=
TypeOfObject::kVariable;
}

bool Node::IsVariable() {
  return std::holds_alternative<TypeOfObject>(object_) &&
         std::get<TypeOfObject>(object_) ==
TypeOfObject::kVariable;
}

bool Node::IsNumber() {
  return std::holds_alternative<long long>(object_);
}
```

my_exception.h

```
#ifndef MY_EXCEPTION_H_
```

```
#define MY_EXCEPTION_H_

#include <iostream>

enum class ErrorCode {
  kNone = 0,
  kIndexError,
  kValueError,
  kSyntaxError,
  kRuntimeError
};

class MyException {
 public:
  MyException(const ErrorCode code, const std::string&& str = "");
  MyException(const MyException& err);
  ErrorCode GetCode();
  friend std::ostream& operator<<(std::ostream& out, const
MyException object);
  ~MyException() = default;

 private:
  ErrorCode code_;
  std::string str_;
};

#endif  // MY_EXCEPTION_H_
```

my_exception.cpp

```
#include "../lib/my_exception.h"

MyException::MyException(const ErrorCode code, const std::string&&
str)
    : code_(code), str_(str) {}

MyException::MyException(const MyException& err)
    : code_(err.code_), str_(err.str_) {}

ErrorCode MyException::GetCode() { return code_; }

std::ostream& operator<<(std::ostream& out, const MyException
object) {
  switch (object.code_) {
    case ErrorCode::kNone:
      out << "None";
```

```
      break;
    case ErrorCode::kIndexError:
      out << "IndexError: " << object.str_;
      break;
    case ErrorCode::kValueError:
      out << "ValueError: " << object.str_;
      break;
    case ErrorCode::kSyntaxError:
      out << "SyntaxError: " << object.str_;
      break;
    case ErrorCode::kRuntimeError:
      out << "RuntimeError: " << object.str_;
      break;
  }

  return out;
}
```

Makefile

```
CXX = g++
TARGET = lab2
CXXFLAGS = -c -std=c++17
CXXOBJFLAGS = -std=c++17
LIBDIR = source/lib
SRCDIR = source/src
SRCS = $(wildcard $(SRCDIR)/*.cpp)
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
     $(CXX) $(CXXOBJFLAGS) $(OBJS) -o $(TARGET)

%.o: $(SRCDIR)/%.cpp $(LIBDIR)/*.h
     $(CXX) $(CXXFLAGS) $<

clean:
     rm -rf $(SRCDIR)/*.o $(TARGET)
```