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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсия

Студент гр. 9304 _____ Арутюнян В.В.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с рекурсивным подходом решения задач, получить навыки программирования рекурсивных процедур и функций на языке программирования C++.

Постановка задачи.

Вариант 22.

Построить синтаксический анализатор для определённого далее понятия логическое_выражение.

логическое_выражение ::= TRUE | FALSE | идентификатор | NOT (операнд) | операция (операнды)

идентификатор::=буква операция::= AND | OR

операнды::= операнд | операнд, операнды

операнд::= логическое_выражение

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:

./lab1 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 LogicalExpression.

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

1. *expr_* – анализируем строка;
2. index_ – индекс, содержащий номер обрабатываемого символа в данный момент;

3. *error_* – ошибка, которая могла возникнуть в ходе работы анализатора, изначально устанавливается в состояние “без ошибок”;
4. *names_* – имена логических выражений в виде строк.

Для начала анализа теста, достаточно создать экземпляр класса, который будет содержать входную строку, но без пробельных символов (удаляются методом *DeleteSpaces()*), и вызвать метод *Analyze()*. В данном методе происходит вызов метода *CheckNextExpr()* и обработка возможных ошибок анализа строки. Последнее происходит с помощью еще одного реализованного класса *MyException()*.

Внутри *CheckNextExpr()* проверяется следующее логическое выражение при помощи нижеприведенных методов класса:

1. *isFalse()* – проверка на “FALSE”;
2. *isTrue()* – проверка на “TRUE”;
3. *isID()* – проверка на идентификатор (букву);
4. *isAnd()* – проверка на “AND”;
5. *isOr()* – проверка на “OR”;
6. *isNot()* – проверка на “NOT”.

В их основе лежит метод класса *CompareStrings()*, который сравнивает строки “FALSE”, “TRUE”, “AND”, “OR”, “NOT”, с подстрокой той же длины в анализируемой строке по нужному индексу. И в зависимости от проверки, которая окажется удачной, происходит либо вызов метода *CheckBool()*, проверяющий верно ли описана операция (*AND, OR, NOT*), либо бездействие (при *FALSE, TRUE, ID*), либо генерация исключения (если ни один из шести вышеперечисленных методов проверки следующего логического выражения не сработал).

В *CheckBool()* совершается несколько простых последовательных действий:

1. Проверяется наличие открывающейся скобки, в случае отрицательного результата генерируется исключение;

2. Вызывается *CheckNextExpr()* для проверки первого аргумента в скобке (*<операция>(<первый>, <второй>)*);
3. Проверяется наличие запятой, в случае отрицательного результата генерируется исключение;
4. Вызывается *CheckNextExpr()* для проверки второго аргумента в скобке (*<операция>(<первый>, <второй>)*);
5. Проверяется наличие закрывающейся скобки, в случае отрицательного результата генерируется исключение.

При обработке операции *NOT* шаги *№3* и *№4* пропускаются.

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 – *GetError()* и перегружен оператор вывода.

enum class ErrorCode необходим для понятной записи кода ошибки.

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

Тестирование.

Программу можно собрать через *Makefile* командой *make*, после этого создается исполняемый файл *lab1*. Существует несколько вариантов провести тестирование:

1. Вызвать *lab1*, указав путь до файла с тестом. (*lab1 path*);
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	AND(OR(A, B), NOT(A)	FALSE SyntaxError: missing ")"
2	AND(OR(A, B), NOT(A))	TRUE
3	WORD	FALSE SyntaxError: invalid syntax
4	AND	FALSE SyntaxError: missing "("
5	FALSE	TRUE
6	AND(OR(A, B) NOT(A))	FALSE SyntaxError: missing ","
7	AND(OR(A, B, NOT(A))	FALSE SyntaxError: missing ")"
8	AND(O(A, B), NOT(A))	FALSE SyntaxError: missing ","
9	AND(OR(, B), NOT(A))	FALSE SyntaxError: invalid syntax
10	AND (OR (A , B),NOT (A))	TRUE
11	NOT(A, AND (OR(OR))	FALSE SyntaxError: missing ")"
12	OR(A, AND(B, OR(C, AND(OR(OR(A, TRUE), AND(NOT(OR(FALSE, d)), FALSE)),TRUE))))	TRUE

Выводы.

В ходе выполнения лабораторной работы были изучены приемы рекурсивного программирования и реализован синтаксический анализатор логического выражения. Использование рекурсии оправдывается более логичной для человека структурой кода и краткой записью.

ПРИЛОЖЕНИЕ А

main.cpp

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

#include "../lib/logical_expression.h"

void PrintAnalyzeResult(LogicalExpression& expr) {
  if (expr.Analyze()) {
    std::cout << "TRUE\n";
  } else {
    std::cout << "FALSE\n" << expr.GetError() << '\n';
  }
}

int main(int argc, char** argv) {
  if (argc == 1) {
    std::cout << "Too small arguments.\n"
              << "example: ./lab1 -s \"AND(A,B)\"
Tests/test/test1.txt\n"
              << "A logical expression is expected after \"-s\"
flag, "
              << "otherwise a file path is expected.\n";
  } else {
    bool is_string = false;

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

      if (is_string) {
        is_string = false;
        LogicalExpression expr = arg;
        PrintAnalyzeResult(expr);

      } 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);
          LogicalExpression expr = str;
```

```
          PrintAnalyzeResult(expr);
          file_in.close();

        } else {
          std::cout << "Couldn't open the file.\n";
        }
      }
    }
  }
  return 0;
}
```

logical_expression.h

```
#ifndef LOGICAL_EXPRESSION_H_
#define LOGICAL_EXPRESSION_H_

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

#include "my_exception.h"

enum class NamesType { kFalse = 0, kTrue, kAnd, kOr, kNot };

class LogicalExpression {
 public:
  LogicalExpression(const std::string& expr);
  bool Analyze();
  MyException GetError();
  ~LogicalExpression() = default;

 private:
  const std::string names_[5] = {"FALSE", "TRUE", "AND", "OR",
"NOT"};
  MyException error_;
  std::string expr_;
  int index_;

  void DeleteSpaces(std::string& str);
  bool CompareStrings(NamesType type);
  bool IsFalse();
  bool IsTrue();
  bool IsID();
  bool IsAnd();
  bool IsOr();
```

```
  bool IsNot();
  void CheckBool(bool is_not = false);
  void CheckNextExpr();
};

#endif  // LOGICAL_EXPRESSION_H_
```

logical_expression.cpp

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

LogicalExpression::LogicalExpression(const std::string &expr)
    : error_(ErrorCode::kNone), expr_(expr), index_(0) {
  DeleteSpaces(expr_);
}

bool LogicalExpression::Analyze() {
  try {
    CheckNextExpr();
    if (index_ != expr_.size()) {
      throw MyException(ErrorCode::kSyntaxError, "invalid
syntax");
    }
  } catch (MyException &err) {
    error_ = err;
    return false;
  }
  return true;
}

void LogicalExpression::CheckNextExpr() {
  if (IsAnd() || IsOr()) {
    CheckBool();
  } else if (IsNot()) {
    CheckBool(true);
  } else if (!(IsFalse() || IsTrue() || IsID())) {
    throw MyException(ErrorCode::kSyntaxError, "invalid syntax");
  }
}

MyException LogicalExpression::GetError() { return error_; }

bool LogicalExpression::IsFalse() { return
CompareStrings(NamesType::kFalse); }
```

```
bool LogicalExpression::IsTrue() { return
CompareStrings(NamesType::kTrue); }

bool LogicalExpression::IsID() {
  char c = expr_[index_];
  bool result = 'a' <= c && c <= 'z' || 'A' <= c && c <= 'Z';
  if (result) ++index_;
  return result;
}

bool LogicalExpression::IsAnd() { return
CompareStrings(NamesType::kAnd); }

bool LogicalExpression::IsOr() { return
CompareStrings(NamesType::kOr); }

bool LogicalExpression::IsNot() { return
CompareStrings(NamesType::kNot); }

void LogicalExpression::CheckBool(bool is_not) {
  if (expr_[index_] != '(') {
    throw MyException(ErrorCode::kSyntaxError, "missing \"(\"");
  }

  ++index_;
  CheckNextExpr();

  if (!is_not) {
    if (expr_[index_] != ',') {
      throw MyException(ErrorCode::kSyntaxError, "missing \",\"");
    }
    ++index_;
    CheckNextExpr();
  }

  if (expr_[index_] != ')') {
    throw MyException(ErrorCode::kSyntaxError, "missing \")\"");
  }
  ++index_;
}

void LogicalExpression::DeleteSpaces(std::string &str) {
  str.erase(std::remove_if(str.begin(), str.end(),
                           [](unsigned char sym) { return
std::isspace(sym); }),
            str.end());
```

```
}

bool LogicalExpression::CompareStrings(NamesType type) {
  bool result =
      !expr_.compare(index_, names_[(int)type].size(),
names_[(int)type]);
  if (result) {
    index_ += names_[(int)type].size();
  }
  return result;
}
```

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 = "");
  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) {}

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

```
CC = g++
TARGET = lab1
CFLAGS = -c
LIBDIR = Source/lib
SRCDIR = Source/src
SRCS = $(wildcard $(SRCDIR)/*.cpp)
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
     $(CC) $(OBJS)  -o $(TARGET)

%.o: $(SRCDIR)/%.cpp $(LIBDIR)/*.h
     $(CC) $(CFLAGS) $<
```

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