

**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 и научиться применять рекурсию в программировании.

## 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 принимает путь до файла с тестом, который требуется обработать, в виде одной строки.

*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 NameType* 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 ErrorCode* необходим для понятной записи кода ошибки.

Программа выводит “TRUE”, если входная строка являлась логическим выражением, 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 рассмотрены изучены приемы рекурсивного программирования и реализован синтаксический анализатор логического выражения.

## ПРИЛОЖЕНИЕ А

### **main.cpp**

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

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

int main(int argc, char** argv) {
    std::ifstream file_in(argv[1]);

    if (file_in.is_open()) {
        std::string str;
        std::getline(file_in, str);

        LogicalExpression expr = str;

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

        file_in.close();
    } else {
        std::cout << "Couldn't open the file.\nusage: ./lab1 filename\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 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_(kNone), expr_(expr), index_(0) {
    DeleteSpaces(expr_);
}

bool LogicalExpression::Analyze() {
    try {
        CheckNextExpr();
        if (index_ != expr_.size()) {
            throw MyException(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(kSyntaxError, "invalid syntax");
    }
}

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

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

bool LogicalExpression::IsTrue() { return CompareStrings(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(kAnd); }

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

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

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

    ++index_;
    CheckNextExpr();

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

```



```

        ++index_;
        CheckNextExpr();
    }

    if (expr_[index_] != ')') {
        throw MyException(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_[type].size(), names_
[type]);
    if (result) {
        index_ += names_[type].size();
    }
    return result;
}

```

### **my\_exception.h**

```

#ifndef MY_EXCEPTION_H_
#define MY_EXCEPTION_H_

#include <iostream>

enum 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&& ob
ject) {
    switch (object.code_) {
        case kNone:
            out << "None";
            break;
        case kIndexError:
            out << "IndexError: " << object.str_;
            break;
        case kValueError:
            out << "ValueError: " << object.str_;
            break;
        case kSyntaxError:
            out << "SyntaxError: " << object.str_;
            break;
        case 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)
```