МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт информационных технологий

Кафедра «Информационные технологии и компьютерные системы»

ОТЧЕТ

по лабораторной работе №3

по дисциплине “Системное программное обеспечение”

Вариант 4

Выполнил:

ст. гр. ИТ/б-22-6-о Донец Н.О.

Принял:

ассистент Ткаченко К.С.

Севастополь

2024 г.

**Цель работы:**

Изучить способы построения регулярных грамматик и соответствующих им конечных автоматов.

**Задание:**

1. Построить конечный автомат для заданной грамматики (в отчете представить граф и таблицу переходов автомата).
2. Разработать и отладить программу лексического анализатора - препроцессор на основе построенной автоматной модели. Лексический анализатор должен быть оформлен в виде отдельной процедуры (подпрограммы).

Грамматика языка Logic4:

<программа>::=<блок>

<блок>::=<оператор>|<оператор>;< блок >

<оператор>:=<переменная>:=<выражение>

<оператор>:= if <переменная> ? <оператор> : <оператор>

<выражение>::=<фактор>|<выражение>#<фактор>

<фактор>::=<первичное>|<фактор>&<первичное>

<первичное>::=<идент.>|<константа>|(<выражение>)

<константа>::=<целая константа>

<целая константа>::=<число>

<число>::=<цифра>|<число><цифра>

<цифра>::=0|1|2|3|4|5|6|7|8|9

<идент.>::=<буква>|<идент.><буква>

<буква>::=A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

**Ход работы:**

Был построен граф конечного автомата (Рисунок 1).

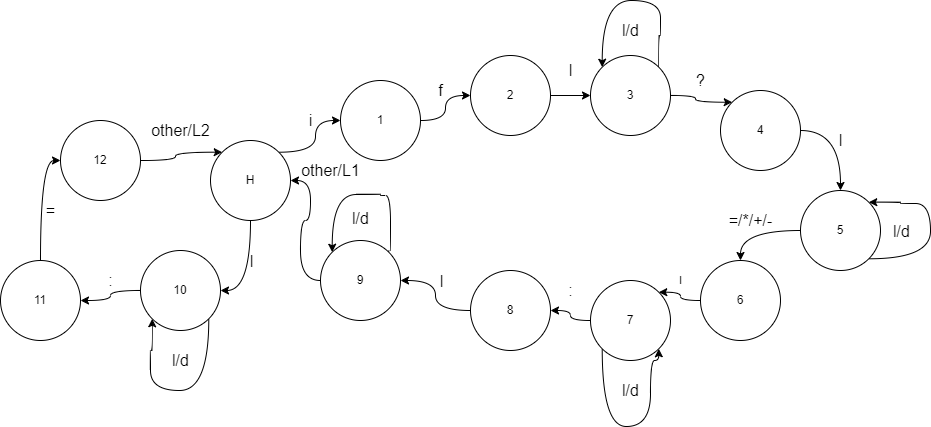


Рисунок 1 – Граф конечного автомата

Была составлена таблица переходов автомата (Таблица 1).

Таблица 1 – Переходы конечного автомата

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | i | f | a..z | 0..9 | ? | = | : | other |
| 0 | 1 |  | 3 |  |  |  |  |  |
| 1 |  | 2 |  |  |  |  |  |  |
| 2 |  |  | 3 |  |  |  |  |  |
| 3 |  |  | 3 | 3 | 4 |  | 10 |  |
| 4 |  |  | 5 |  |  |  |  |  |
| 5 |  |  | 5 | 5 |  | 6 |  |  |
| 6 |  |  | 7 |  |  |  |  |  |
| 7 |  |  | 7 | 7 |  |  | 8 |  |
| 8 |  |  | 9 |  |  |  |  |  |
| 9 |  |  | 9 | 9 |  |  |  | 0/L1 |
| 10 |  |  |  |  |  | 11 |  |  |
| 11 |  |  |  |  |  |  |  | 0/L2 |

Также был разработан класс лексического анализатора (Листинг 1).

Листинг 1 – Лексический анализатор

#ifndef LEXICAL\_ANALYZER

#define LEXICAL\_ANALYZER

#include <cctype>

#include <iostream>

#include <vector>

#include <sstream>

#include <algorithm>

#include <map>

#include <string>

#include <list>

class LexicalAnalyzer {

    private:

      std::vector<std::string> divideIntoSubStrings(std::string str);

      std::list<std::string> lexems;

      std::map<std::string, std::string> keywords;

      std::map<std::string, std::string> operators;

      int validateString(std::string str);

      int validateConstant(std::string str, int start);

      int validateIdentifier(std::string str, int start);

      int validateOperator(std::string str, int start);

      void setKeywords();

      void setOperators();

    public:

      LexicalAnalyzer();

      std::string analyze(std::string str);

};

#endif

#include "LexicalAnalyzer.h"

//a +b aboba=4 if3 if 12-4 6 +5

LexicalAnalyzer::LexicalAnalyzer() {

    setKeywords();

    setOperators();

}

void LexicalAnalyzer::setKeywords() {

    //term

    keywords.emplace("if", "T1");

}

void LexicalAnalyzer::setOperators() {

    //term

    operators.emplace("?", "T2");

    operators.emplace(":", "T3");

    //divide

    operators.emplace("#", "D");

    operators.emplace("&", "D");

    //assigment

    operators.emplace(":=", "As");

    operators.emplace("=", "As");

    //end

    operators.emplace(";", "E");

    //ariphmetic

    operators.emplace("+", "Ar");

    operators.emplace("-", "Ar");

    operators.emplace("\*", "Ar");

    operators.emplace("/", "Ar");

}

std::vector<std::string> LexicalAnalyzer::divideIntoSubStrings(std::string str) {

    std::vector<std::string> subStrings;

    std::istringstream iss(str);

    std::string token;

    while(std::getline(iss, token, ' ')) {

        subStrings.push\_back(token);

    }

    return subStrings;

}

std::string LexicalAnalyzer::analyze(std::string str) {

    std::vector<std::string> subStrings = divideIntoSubStrings(str);

    std::string final;

    for (auto str : subStrings) {

        validateString(str);

    }

    for (auto str : lexems) {

        final += str;

    }

    return final;

}

int LexicalAnalyzer::validateString(std::string str) {

    if (isdigit(str[0])) {

        if(validateConstant(str, 0) == -1) {

            return -1;

        }

        return 0;

    }

    if (keywords.find(str) != keywords.end()) {

        lexems.push\_back(keywords[str]);

        std::cout << "Keyword: " << str <<" lexem: " << keywords[str] << std::endl;

        return 0;

    }

    if (validateIdentifier(str, 0) == -1) {

        return -1;

    }

    return 0;

}

int LexicalAnalyzer::validateConstant(std::string str, int start) {

    std::string token;

    for (int i = start; i < str.size(); i++) {

        if (!isdigit(str[i])) {

            int j;

            if ((j = validateOperator(str, i)) != -1) {

                if (token.size() > 0) {

                    auto position = lexems.end();

                    position--;

                    lexems.insert(position, "C");

                    std::cout << "Constant: " << token << " lexem: C" << std::endl;

                    token = "";

                }

                i=j-1;

                if (isalpha(str[j])) {

                    if(validateConstant(str, j) == -1) {

                        return -1;

                    }

                    return 0;

                }

                continue;

            }

            else {

                if (isalpha(str[i])) {

                    std::cout << "Identifier can't start with the number: " << str << std::endl;

                }

                else {

                    std::cout << "Unexpected symbol: " << str[i] << " in: " << str << std::endl;

                }

                lexems.push\_back(" error ");

                return -1;

            }

        }

        token += str[i];

    }

    if (token.size() > 0) {

        lexems.push\_back("C");

        std::cout << "Constant: " << token << " lexem: C" << std::endl;

    }

    return 0;

}

int LexicalAnalyzer::validateIdentifier(std::string str, int start) {

    std::string token;

    for (int i = start; i < str.size(); i++) {

        if (!isalpha(str[i]) && !isdigit(str[i])) {

            int j;

            if ((j = validateOperator(str, i)) != -1) {

                if (token.size() > 0) {

                    auto position = lexems.end();

                    position--;

                    lexems.insert(position, "I");

                    std::cout << "Identifier: " << token << " lexem: I" << std::endl;

                    token = "";

                }

                i=j-1;

                if (isdigit(str[j])) {

                    if(validateConstant(str, j) == -1) {

                        return -1;

                    }

                    return 0;

                }

                continue;

            }

            else {

                std::cout << "Unexpected symbol: " << str[i] << " in: " << str << std::endl;

                lexems.push\_back(" error ");

                return -1;

            }

        }

        token += str[i];

    }

    if (token.size() > 0) {

        lexems.push\_back("I");

        std::cout << "Identifier: " << token << " lexem: I" << std::endl;

    }

    return 0;

}

int LexicalAnalyzer::validateOperator(std::string str, int start) {

    std::string token;

    int i = start;

    for (; i < str.size(); i++) {

        if (isdigit(str[i]) || isalpha(str[i])) break;

        else token+=str[i];

    }

    if (operators.find(token) != operators.end()) {

        std::cout << "Operator: " << token << " lexem: " << operators[token] << std::endl;

        lexems.push\_back(operators[token]);

        return i;

    }

    else {

        return -1;

    }

}

Была разработана программа для проверки корректности работы анализатора (Листинг 2).

Листинг 2 – Программа для проверки

#include "LexicalAnalyzer.h"

#include <iostream>

#include <string>

#include <windows.h>

int main() {

std::cout << "String to analyze: ";

std::string input;

getline(std::cin, input);

LexicalAnalyzer lexicalAnalyzer;

std::cout << lexicalAnalyzer.analyze(input) << std::endl;

return 0;

}

Также были проведены тесты программы (рисунки 2 – 3).

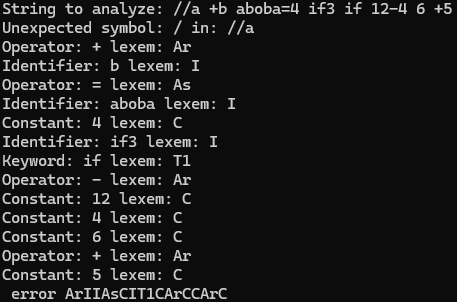


Рисунок 2 – Первый тест

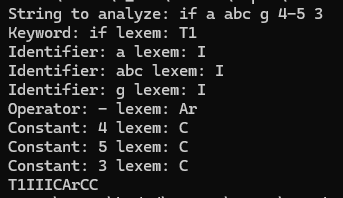


Рисунок 3 – Второй тест

**Выводы**

В ходе лабораторной работы были изучены способы построения регулярных грамматик и соответствующих им конечных автоматов.