НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота №1

з дисципліни
«Основи проектування трансляторів»
Тема: **«Розробка лексичного аналізатора»**

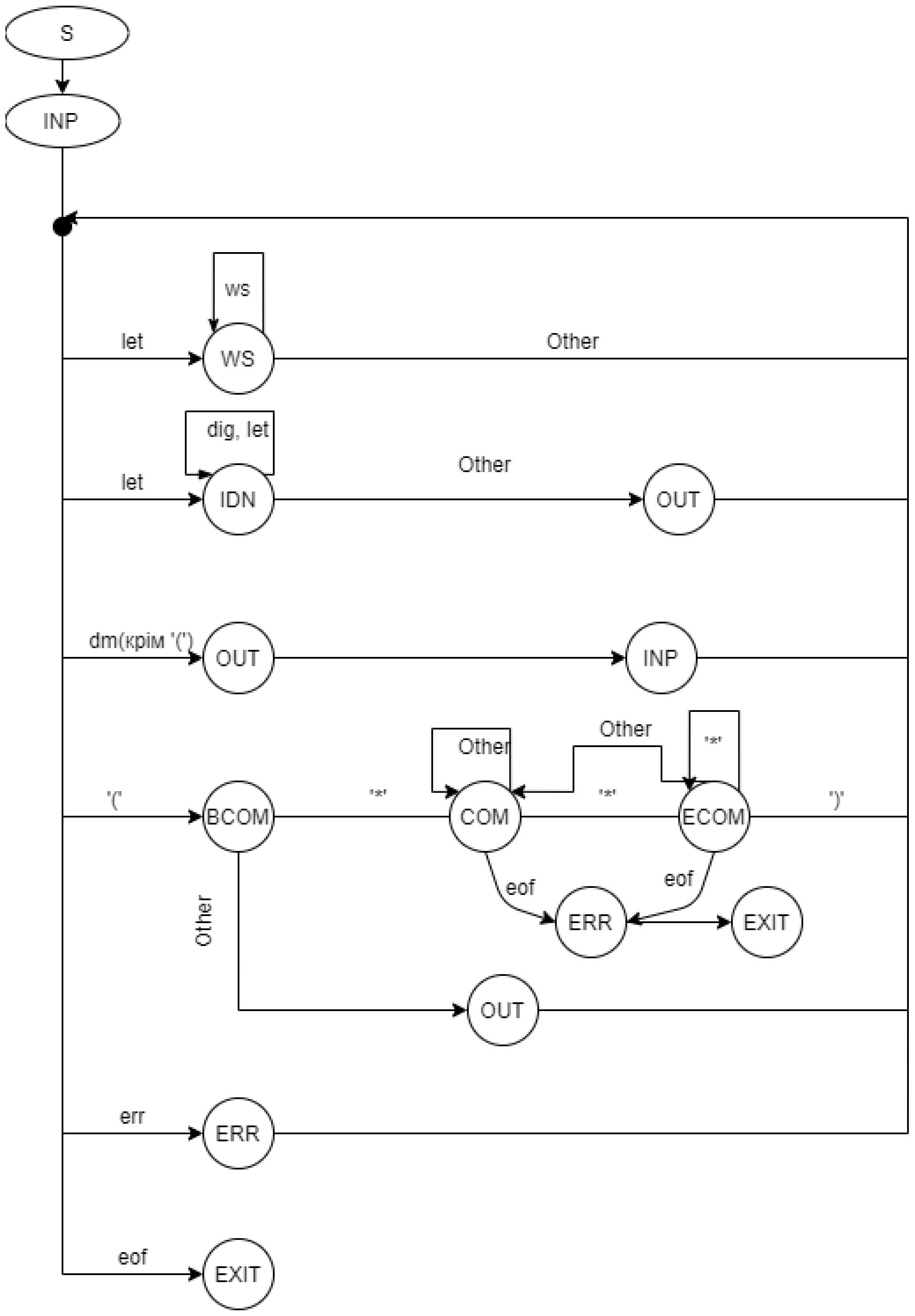
Виконав: студент III курсу
ФПМ групи КВ-82
Бікерей О.І.

Київ 2021

Варіант №2

```
1. <signal-program> --> <program>
2. <program> --> PROGRAM <procedure-identifier> ;
<block>. |
PROCEDURE <procedureidentifier><parameters-list> ; <block> ;
3. <block> --> BEGIN <statements-list> END
4. <statements-list> --> <empty>
5. <parameters-list> --> ( <declarations-list> ) | <empty>
6. <declarations-list> --> <declaration><declarations-list> |
<empty>
7. <declaration> --
><variableidentifier><identifierslist>:<attribute><attributes-list> ;
8. <identifiers-list> --> , <variable-identifier>
<identifiers-list> |
<empty>
9. <attributes-list> --> <attribute> <attributeslist> | <empty>
10. <attribute> --> SIGNAL |
COMPLEX |
INTEGER |
FLOAT |
BLOCKFLOAT |
EXT
11. <procedure-identifier> --> <identifier>
12. <variable-identifier> --> <identifier>
13. <identifier> --> <letter><string>
14. <string> --> <letter><string> | <digit><string> | <empty>
15. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |9
16. <letter> --> A | B | C | D | ... | Z
```

Граф автомату:



Лістинг програми

Main.cpp

```cpp
#include "Table.h"
#include "LexicalAnalyzer.h"

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

using namespace std;

int main(int argc, char **argv) {
    if (argc != 2) {
        cout << "No arguments passed" << endl;
        exit(1);
    }

    string filename = argv[1];
    string directory;
    const size_t last_slash_idx = filename.rfind('\\');
    if (std::string::npos != last_slash_idx) {
        directory = filename.substr(0, last_slash_idx);
    }
    directory.append("\\generated.txt");

    ifstream f(filename);
    ofstream out(directory);

    if (f.is_open()) {
        Table table;
        LexicalAnalyzer lexicalAnalyzer(out, f, table);

        if (!lexicalAnalyzer.start()) {
            cout << "LexicalAnalyzer has been failed" << endl;
        }
    } else {
        cout << "Unable to open file " << (const char *) argv[1];
    }

    f.close();
    out.close();

    return 0;
}
```

CharType.h

```cpp
//
// Created by oleks on 21.03.2021.
//

#ifndef LAB1_CHARTYPE_H
#define LAB1_CHARTYPE_H

enum class CharType {
    DIG,
    LET,
    DM,
    COM,
    WS,
    Eof,
    ERR,
};


#endif //LAB1_CHARTYPE_H
```

IdType.h

```
//
// Created by oleks on 29.03.2021.
//

#ifndef LAB1_IDTYPE_H
#define LAB1_IDTYPE_H

enum class IdType {
    DM,
    Keyword,
    Id
};

#endif //LAB1_IDTYPE_H
```

LexicalAnalyzer.h

```
//
// Created by oleks on 21.03.2021.
//

#ifndef LAB1_LEXICALANALYZER_H
#define LAB1_LEXICALANALYZER_H

#include "CharType.h"
#include "IdType.h"
#include "PropertyLocation.h"
#include "Table.h"

#include <istream>
#include <ostream>
#include <cstdarg>

using namespace std;

class LexicalAnalyzer {
private:
    istream *stream;
    ostream *output;
    Table *tab;
    string buffer;
    CharType type;
    char current;
    int position;
    int col;
    int lines;

    int prevLines;
    int prevCol;

    void setNext();

    void setBuffer();

    void reset();
    int makeId();

    int makeDm();

    PropertyLocation getPosInfo(int id = 0);

    void printError(string format, ...);

public:
    vector<PropertyLocation> lexemes;

    explicit LexicalAnalyzer(ostream &output, istream &stream, Table &table);

    bool start();
};


#endif //LAB1_LEXICALANALYZER_H
```

LexicalAnalyzer.cpp

```
//
// Created by oleks on 21.03.2021.
//
#include "LexicalAnalyzer.h"

void LexicalAnalyzer::setNext() {
    position++;
    char chr;

    prevLines = lines;
    prevCol = col;

    if ((chr = stream->get()) == '\n') {
        col = position;
        lines++;
    }

    current = chr;
    type = tab->getChar(chr);
}

void LexicalAnalyzer::setBuffer() {
    buffer.push_back(current);
}

void LexicalAnalyzer::reset() {
    buffer.clear();
    type = CharType::ERR;
    lexemes.clear();

    current = 0;
    position = 0;
    col = 0;
    lines = 0;
}


int LexicalAnalyzer::makeId() {
    return tab->makeId(buffer);
}

int LexicalAnalyzer::makeDm() {
    return tab->makeDm(current);
```

```
}


PropertyLocation LexicalAnalyzer::getPosInfo(int id) {
    return PropertyLocation{id,
                            prevLines + 1,
                            position - prevCol - (int) buffer.size()};
}

void LexicalAnalyzer::printError(string format, ...) {
    char buff[300];

    va_list argp;
    va_start(argp, format);
    vsnprintf(buff, sizeof(buff), format.c_str(), argp);
    va_end(argp);

    *output << "LexicalAnalyzer: Error ( line: " << prevLines + 1 << ",
column " << position - 1 - prevCol << " ): ";
    *output << (const char *) buff << endl;
}

LexicalAnalyzer::LexicalAnalyzer(ostream &output, istream &stream, Table
&table) {
    this->stream = &stream;
    this->output = &output;
    this->tab = &table;
    reset();
}

bool LexicalAnalyzer::start() {
    reset();
    setNext();
    bool exit = false;
    bool abort = false;
    while (!exit) {
        buffer.clear();
        switch (type) {
            case CharType::WS:
                while (type == CharType::WS)
                    setNext();
                break;
            case CharType::LET:
                while (type == CharType::DIG || type == CharType::LET) {
                    setBuffer();
                    setNext();
                }
                lexemes.push_back(getPosInfo(makeId()));
                break;
            case CharType::DM:
                lexemes.push_back(getPosInfo(makeDm()));
                setNext();
                break;
            case CharType::COM:
                setNext();
                if (current == '*') {
                    bool comment = true;
                    setNext();
                    while (comment) {
                        while (current != '*') {
                            if (type == CharType::Eof) {
                                printError("File ended before comment was
closed", current);
                                return false;
                            }
                            setNext();
                        }
                        setNext();
                        if (current == ')')
                            comment = false;
                    }
                    setNext();
                } else {
                    lexemes.push_back(getPosInfo('('));
                    setNext();
                }
                break;
            case CharType::Eof:
                exit = true;
                break;
            case CharType::ERR:
                printError("Illegal character `%c` detected", current);
                abort = true;
                setNext();
                break;
            default:
                return false;
        }
    }

    if(!abort) {
        for (auto iter : lexemes) {
            iter.print(output, tab);
        }
    }
    return !abort;
}
```

PropertyLocation.h

```
//
// Created by oleks on 21.03.2021.
//

#ifndef LAB1_PROPERTYLOCATION_H
#define LAB1_PROPERTYLOCATION_H

#include "Table.h"
#include <ostream>
#include <iomanip>
#include <string>

using namespace std;

class PropertyLocation {
public:
    int id;
    int line;
    int column;

    PropertyLocation(int id, int line, int column);

    void print(ostream *stream, Table *tab) const;
};


#endif //LAB1_PROPERTYLOCATION_H
```

PropertyLocation.cpp

```cpp
//
// Created by oleks on 21.03.2021.
//

#include"PropertyLocation.h"


PropertyLocation::PropertyLocation(int id, int line, int column) {
    this->id = id;
    this->line = line;
    this->column = column;
}

void PropertyLocation::print(ostream *stream, Table *tab) const {
    string val;
    switch (tab->classifyIndex(id)) {
        case IdType::DM:
            val = (char) id;
            break;
        case IdType::Keyword:
            val = tab->getKeyword(id);
            break;
        case IdType::Id:
            val = tab->getId(id);
            break;
        default:
            val = "ERR";
    }
    *stream << setw(3) << line << " | "
            << setw(3) << column << " | "
            << setw(7) << id << " | "
            << val << endl;
}
```

Table.h

```cpp
//
// Created by oleks on 21.03.2021.
//

#ifndef LAB1_TABLE_H
#define LAB1_TABLE_H

#include "CharType.h"
#include "IdType.h"

#include <string>
#include <vector>
#include <map>
#include <algorithm>

using namespace std;

class Table {
private:
    map<char, CharType> chars;
    vector<string> keywords;
    vector<string> ids;

    const int offsetChar = 0;
    const int offsetDM = 256;
    const int offsetKeyword = 400;
    const int offsetId = 1000;

    void setupChars();

    void setupKeywords();

public:
    Table();

    int makeId(string &buffer);

    int makeDm(char chr);

    CharType getChar(char chr) const;

    string getKeyword(int id) const;

    string getId(int id) const;


    IdType classifyIndex(int id) const;
};

#endif //LAB1_TABLE_H
```

Table.cpp

```
//
// Created by oleks on 21.03.2021.
//
#include "Table.h"

using namespace std;

void Table::setupChars() {
    for (int i = 0; i < 255; i++)
        chars[i] = CharType::ERR;

    for (int i = 8; i < 15; i++)
        chars[i] = CharType::WS; //tab \r \t etc.

    for (int i = 48; i < 58; i++)
        chars[i] = CharType::DIG; // 0 1 2 3 4 ...


    for (int i = 65; i < 91; i++)
        chars[i] = CharType::LET; // A B C D ...

    chars[32] = CharType::WS;   // space
    chars[40] = CharType::COM;   // (
    chars[41] = CharType::DM; // )
    chars[58] = CharType::DM; // :
    chars[46] = CharType::DM; // .
    chars[59] = CharType::DM; // ;
    chars[44] = CharType::DM; // ,

    chars[EOF] = CharType::Eof;
}
```

```
void Table::setupKeywords() {
    keywords.emplace_back("PROGRAM");
    keywords.emplace_back("PROCEDURE");
    keywords.emplace_back("BEGIN");
    keywords.emplace_back("END");
    keywords.emplace_back("SIGNAL");
    keywords.emplace_back("COMPLEX");
    keywords.emplace_back("INTEGER");
    keywords.emplace_back("FLOAT");
    keywords.emplace_back("BLOCKFLOAT");
    keywords.emplace_back("EXT");
}


Table::Table() {
    setupChars();
    setupKeywords();
}


int Table::makeId(string &buffer) {
    auto iter = find(keywords.begin(), keywords.end(), buffer);
    if (iter != keywords.end()) {
        return (int) distance(keywords.begin(), iter) + offsetKeyword;
    } else {
        auto iter = find(ids.begin(), ids.end(), buffer);
        if (iter != ids.end()) {
            return (int) distance(ids.begin(), iter) + offsetId;
        } else {
            ids.push_back(buffer);
            return (int) ids.size() - 1 + offsetId;
        }

    }
}

int Table::makeDm(char chr) {
    return chr;
}


CharType Table::getChar(char chr) const {
    return chars.at(chr);
}

string Table::getKeyword(int id) const {
    return keywords.at(id - offsetKeyword);
}

string Table::getId(int id) const {
    return ids.at(id - offsetId);
}

IdType Table::classifyIndex(int id) const {
    if (id > offsetChar && id < offsetDM) {
        if (chars.at(id) == CharType::DM || chars.at(id) == CharType::COM)
            return IdType::DM;
        else
            abort();
    } else if (id < offsetId) return IdType::Keyword;
    else return IdType::Id;
}
```