НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО» ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Лабораторна робота №1

з дисципліни «Основи проектування трансляторів»

Тема: «РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА»

Виконав: студент III курсу

ФПМ групи КВ-81

Ядуха Б.В.

Викладач: Марченко О. І.

Київ 2021

Мета лабораторної роботи

Метою лабораторної роботи «Розробка лексичного аналізатора» є засвоєння теоретичного матеріалу та набуття практичного досвіду і практичних навичок розробки лексичних аналізаторів (сканерів).

Варіант 19

1. <signal-program> --> <program>

2. <program> --> PROGRAM <procedure-identifier> ;
<block>.

3. <block> --> <variable-declarations> BEGIN
<statements-list> END

4. <variable-declarations> --> VAR <declarations-list> |
<empty>

5. <declarations-list> --> <declaration>
<declarations-list> |
<empty>

6. <declaration> --> <variable-identifier>:<attribute> ;

7. <attribute> --> INTEGER |
FLOAT

8. <statements-list> --> <statement> <statementslist> |
<empty>

9. <statement> --> <condition-statement> ENDIF ;

10. <condition-statement> --> <incompletecondition-statement><alternative-part>

11. <incomplete-condition-statement> --> IF
<conditional-expression> THEN
<statements-list>

12. <alternative-part> --> ELSE <statements-list> |
<empty>

13. <conditional-expression> --> <expression> = <expression>

14. <expression> --> <variable-identifier> |
<unsigned-integer>

15. <variable-identifier> --> <identifier>

16. <procedure-identifier> --> <identifier>

17. <identifier> --> <letter><string>

18. <string> --> <letter><string> |
<digit><string> |
<empty>

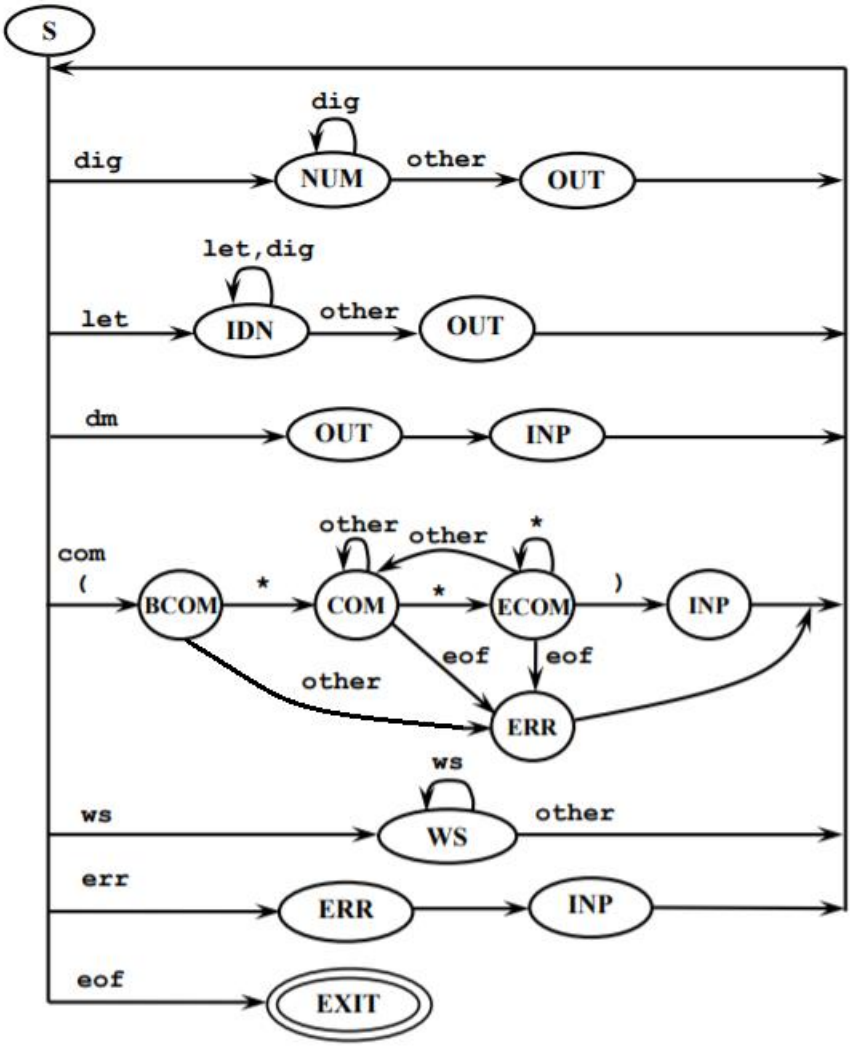
19. <unsigned-integer> --> <digit><digits-string>

20. <digits-string> --> <digit><digits-string> |
<empty>

21. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

22. <letter> --> A | B | C | D | ... | Z

Граф автомату



Source.cpp

```
#include "lexer.h"

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char** args)
{
	fill_symbol_categories();
	for (int i = 0; i < argc; i++)
	{
		string path(args[i]);
		Lexer l;
		l.Parse(path + "/test.sig");
		ofstream of(path + "/generated.txt");
		l.Print(of);
		l.ShowErrors(of);
		of.close();
	}
}
```

symbol.h

```
#pragma once

#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

enum class SymbolCategories
{
	WhiteSpace,
	Number,
	Leter,
	Delimiter,
	CommentDelimiter,
	InvalidCharacters,
	Eof
};

extern vector<SymbolCategories> symbol_categories;


void fill_symbol_categories();

class Symbol
{
public:
	char GetValue() const;
	SymbolCategories GetType() const;
	bool get(ifstream& is);
private:
	char value;
	SymbolCategories type;
};
```

symbol.cpp

```cpp
#include "symbol.h"

#include <cctype>

using namespace std;

vector<SymbolCategories> symbol_categories(256, SymbolCategories::InvalidCharacters);

void fill_symbol_categories()
{
	symbol_categories[32] = SymbolCategories::WhiteSpace;
	for (int i = 8; i < 14; i++)
	{
		symbol_categories[i] = SymbolCategories::WhiteSpace;
	}
	for (int i = 48; i < 58; i++)
	{
		symbol_categories[i] = SymbolCategories::Number;
	}
	for (int i = 97; i < 123; i++)
	{
		symbol_categories[i] = SymbolCategories::Leter;
	}
	for (int i = 65; i < 91; i++)
	{
		symbol_categories[i] = SymbolCategories::Leter;
	}
	symbol_categories[58] = SymbolCategories::Delimiter;
	symbol_categories[59] = SymbolCategories::Delimiter;
	symbol_categories[61] = SymbolCategories::Delimiter;
	symbol_categories[40] = SymbolCategories::CommentDelimiter;
}

char Symbol::GetValue() const
{
	return value;
}

SymbolCategories Symbol::GetType() const
{
	return type;
}

bool Symbol::get(ifstream& is)
{

	if (is.get(value))
	{
		type = symbol_categories[value];
		return true;
	}
	type = SymbolCategories::Eof;
	return false;
}
```

lexer.h

```cpp
#pragma once
#include "tables.h"
#include "symbol.h"

#include <string>

using namespace std;

class Lexer
{
public:
	void Parse(string input_file_name);
	void Print(ofstream& os);
	void ShowErrors(ofstream& os);
private:
	struct tocken_table_field
	{
		size_t column, line, id;
		string value;
	};
	vector<tocken_table_field> tocken_table;
	vector<string> errors;
};
```

lexer.cpp

```
#include "lexer.h"

#include <fstream>

using namespace std;

map<string, size_t> key_word_table = { {"PROGRAM", 401}, {"BEGIN", 402}, {"END", 403},
                                                                        {"VAR", 404},
{"INTEGER", 405}, {"FLOAT", 406},
                                                                        {"ENFIF", 407},
{"IF", 408}, {"THEN", 409}, {"ELSE", 410}};
map<string, size_t> identifier_table;
map<string, size_t> constants_table;

void Lexer::Parse(string input_file_name)
{
	tocken_table.clear();
	errors.clear();
	ifstream input_file(input_file_name);
	Symbol current_symbol;
	if (current_symbol.get(input_file))
	{
		size_t constants_id = 501, identifier_id = 1001;
		size_t column = 1, line = 1, bcolumn = 1, bline = 1;
		string current_tocken = "";
		while (current_symbol.GetType() != SymbolCategories::Eof)
		{
			current_tocken = "";
			switch (current_symbol.GetType())
			{
			case SymbolCategories::Number:
				while (current_symbol.GetType() ==
SymbolCategories::Number)
				{
					current_tocken += current_symbol.GetValue();
					current_symbol.get(input_file);
					column++;
				}
				{
					auto it = constants_table.find(current_tocken);
					if (it != constants_table.end())
					{
						tocken_table.push_back({ bcolumn, bline,
(*it).second, current_tocken });
					}
					else
					{
						tocken_table.push_back({bcolumn, bline,
constants_id, current_tocken });
						constants_table[current_tocken] =
constants_id;
						constants_id++;
					}
				}
				bcolumn = column;
				break;
			case SymbolCategories::Leter:
				while (current_symbol.GetType() ==
SymbolCategories::Number || current_symbol.GetType() == SymbolCategories::Leter)
				{
					current_tocken += current_symbol.GetValue();
```

```
                                        current_symbol.get(input_file);
                                        column++;
                                }
                                {
                                        auto it = key_word_table.find(current_tocken);
                                        if (it != key_word_table.end())
                                        {
                                                tocken_table.push_back({ bcolumn, bline,
(*it).second, current_tocken });
                                        }
                                        else
                                        {
                                                it = identifier_table.find(current_tocken);
                                                if (it != identifier_table.end())
                                                {
                                                        tocken_table.push_back({ bcolumn,
bline, (*it).second, current_tocken });
                                                }
                                                else
                                                {
                                                        tocken_table.push_back({ bcolumn,
bline, identifier_id, current_tocken });
                                                        identifier_table[current_tocken] =
identifier_id;
                                                        identifier_id++;
                                                }
                                        }
                                }
                                bcolumn = column;
                                break;
                        case SymbolCategories::Delimiter:
                                {
                                        current_tocken += current_symbol.GetValue();
                                        tocken_table.push_back({ bcolumn, bline,
(size_t)current_symbol.GetValue(), current_tocken });
                                }
                                column++;
                                bcolumn = column;
                                current_symbol.get(input_file);
                                break;
                        case SymbolCategories::CommentDelimiter:
                                column++;
                                if (current_symbol.get(input_file) &&
current_symbol.GetValue() == '*')
                                {
                                        column++;
                                        while (true)
                                        {
                                                if (current_symbol.GetType() ==
SymbolCategories::Eof)
                                                {
                                                        errors.push_back("Lexer: Error(line "
+ to_string(bline) + ", column " + to_string(bcolumn) + "): Not a closed comment");
                                                        break;
                                                }
                                                else
                                                {
                                                        if (current_symbol.GetValue() == '*')
                                                        {
                                                                current_symbol.get(input_file);
                                                                column++;
```

```
                                                        if (current_symbol.GetType() !=
SymbolCategories::Eof && current_symbol.GetValue() == ')')
                                                        {

        current_symbol.get(input_file);
                                                                break;
                                                        }
                                                }
                                                else
                                                {
                                                        if (current_symbol.GetValue()
== '\n')
                                                        {
                                                                column = 1;
                                                                line++;
                                                        }
                                                        current_symbol.get(input_file);
                                                        column++;
                                                }
                                        }
                                }
                        }
                        else
                        {
                                errors.push_back("Lexer: Error(line " +
to_string(bline) + ", column " + to_string(bcolumn) + "): Expected comment");
                        }
                        bcolumn = column;
                        bline = line;
                        break;
                case SymbolCategories::WhiteSpace:
                        while (current_symbol.GetType() ==
SymbolCategories::WhiteSpace)
                        {
                                if (current_symbol.GetValue() == '\n')
                                {
                                        column = 1;
                                        line++;
                                }
                                else
                                {
                                        column++;
                                }
                                current_symbol.get(input_file);
                        }
                        bcolumn = column;
                        bline = line;
                        break;
                case SymbolCategories::InvalidCharacters:
                        errors.push_back("Lexer: Error(line " + to_string(line) +
", column " + to_string(column) +  "): Invalid character: '" +
current_symbol.GetValue() + '\'');
                        current_symbol.get(input_file);
                        column++;
                        break;
                }
        }
        }
        else
        {
                errors.push_back("Lexer: Error: Empty file");
        }
```

```
	input_file.close();
}


void Lexer::Print(ofstream& os)
{
	for (const tocken_table_field& elem : tocken_table)
	{
		os << elem.line << " " << elem.column << " " << elem.id << " " <<
elem.value << '\n';
	}
}

void Lexer::ShowErrors(ofstream& os)
{
	for (const auto& elem : errors)
	{
		os << elem << '\n';
	}
}
```

tables.h

```
#pragma once
#include <map>
#include <set>
#include <string>
#include <fstream>

using namespace std;



extern map<string, size_t> key_word_table;
extern map<string, size_t> identifier_table;
extern map<string, size_t> constants_table;
```

Контрольні приклади

Приклад №1

input.sig:

```
PROGRAM PR;
BEGIN
	(*ada**)
	VAR v1:FLOAT v2:INTEGER;(*sasa*)
	IF 1 = 2 THEN
		1212;
	ELSE
		213 ;
	ENDIF;
1212;
END
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
4 2 404 VAR
4 6 1002 v1
4 8 58 :
4 9 406 FLOAT
4 15 1003 v2
4 17 58 :
4 18 405 INTEGER
4 25 59 ;
5 2 408 IF
5 5 501 1
5 7 61 =
5 9 502 2
5 11 409 THEN
6 3 503 1212
6 7 59 ;
7 2 410 ELSE
8 3 504 213
```

8 7 59 ;
9 2 1004 ENDIF
9 7 59 ;
10 1 503 1212
10 5 59 ;
11 1 403 END

Приклад №2

input.sig:

```
PROGRAM PR;
BEGIN
        IF VAR <> 12 THEN
        1212;
        ELSE
        213 ;
1212;
END
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
3 2 408 IF
3 5 404 VAR
3 12 501 12
3 15 409 THEN
4 2 503 1212
4 6 59 ;
5 2 410 ELSE
6 2 504 213
6 6 59 ;
7 1 503 1212
7 5 59 ;
8 1 403 END
Lexer: Error(line 3, column 9): Invalid character: '<'
Lexer: Error(line 3, column 10): Invalid character: '>'
```

Приклад №3

input.sig:

```
PROGRAM IDENTIF01;
(*PROCEDURE IDENTIF02 (VAR01,VAR02: INTEGER,FLOAT);

BEGIN
PROGRAM PR;
BEGIN
        (*
*s***
*ssd*
ds**s**)
        (******)IF 1= 1 THEN(*s*s*)
        (**)1212;
        (*s*--==``s*)ELSE
        (*3*2*3*)213 ;
1212;
END
(**)
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
7 10 408 IF
7 13 501 1
7 14 61 =
7 16 501 1
7 18 409 THEN
8 6 503 1212
8 10 59 ;
9 15 410 ELSE
10 11 504 213
10 15 59 ;
11 1 503 1212
11 5 59 ;
```

```
12 1 403 END
```

Приклад №4

input.sig:

```
PROGRAM PR;
BEGIN
	(*ada
	VAR v1:FLOAT v2:INTEGER;(*sasa*)
	IF 1 = 2 THEN
		1212;
	ELSE
		213 ;
	ENDIF;
1212;
END
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
5 2 408 IF
5 5 501 1
5 7 61 =
5 9 502 2
5 11 409 THEN
6 3 503 1212
6 7 59 ;
7 2 410 ELSE
8 3 504 213
8 7 59 ;
9 2 1004 ENDIF
9 7 59 ;
10 1 503 1212
10 5 59 ;
11 1 403 END
```

Приклад №5

input.sig:

```
PROGRAM PR;
```

```
BEGIN
        (*ada**)
        VAR v1:FLOAT v2:INTEGER;(
(
        IF 1 = 2 THEN
                1212;
        ELSE
                213 ;
(*      ENDIF;
1212;
END
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
4 2 404 VAR
4 6 1002 v1
4 8 58 :
4 9 406 FLOAT
4 15 1003 v2
4 17 58 :
4 18 405 INTEGER
4 25 59 ;
6 2 408 IF
6 5 501 1
6 7 61 =
6 9 502 2
6 11 409 THEN
7 3 503 1212
7 7 59 ;
8 2 410 ELSE
9 3 504 213
9 7 59 ;
Lexer: Error(line 4, column 26): Expected comment
Lexer: Error(line 5, column 1): Expected comment
Lexer: Error(line 10, column 1): Not a closed comment
```

Приклад №6

input.sig:

```
PROGRAM PR;
BEGIN

	VAR v1:FLOAT v2:INTEGER;
*)
	IF 1 = 2 THEN
		1212;
	ELSE
		213 ;
	ENDIF;
1212;
END
```

expected.txt:

```
1 1 401 PROGRAM
1 9 1001 PR
1 11 59 ;
2 1 402 BEGIN
4 2 404 VAR
4 6 1002 v1
4 8 58 :
4 9 406 FLOAT
4 15 1003 v2
4 17 58 :
4 18 405 INTEGER
4 25 59 ;
6 2 408 IF
6 5 501 1
6 7 61 =
6 9 502 2
6 11 409 THEN
7 3 503 1212
7 7 59 ;
8 2 410 ELSE
9 3 504 213
9 7 59 ;
```

10 2 1004 ENDIF
10 7 59 ;
11 1 503 1212
11 5 59 ;
12 1 403 END
Lexer: Error(line 5, column 1): Invalid character: '*'
Lexer: Error(line 5, column 2): Invalid character: ')'