Міністерство освіти і науки України

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського» Факультет прикладної математики Кафедра системного програмування і спеціалізованих комп'ютерних системи

Лабораторна робота №2

«РОЗРОБКА ГЕНЕРАТОРА КОДУ»
з дисципліни
«ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ»

Виконав:

студент групи КВ-83

Лазуткин.О.О

Перевірив:

Северін С.

Київ – 2021

Загальне завдання

Розробити програму синтаксичного аналізатора (СА) для підмножини мови програмування SIGNAL згідно граматики за варіантом.

Варіант 11

Граматика підмножини мови програмування SIGNAL:

Варіант 11

```
1.    <signal-program> --> <program>
2.    <program> --> PROGRAM <procedure-identifier> ;
                    <block>.
3.    <block> --> <declarations> BEGIN <statements-
                    list> END
4.    <declarations> --> <label-declarations>
5.    <label-declarations> --> LABEL <unsigned-
                    integer> <labels-list>; |
                    <empty>
6.    <labels-list> -->  , <unsigned-integer>
                    <labels-list> |
                    <empty>
7.    <statements-list> --> <statement> <statements-
                    list> |
                    <empty>
8.    <statement> --> <unsigned-integer> :
                    <statement> |
                    GOTO <unsigned-integer> ; |
                    LINK <variable-identifier> , <unsigned-
                    integer> ; |
                    IN <unsigned-integer>;  |
                    OUT <unsigned-integer>;
9.    <variable-identifier> --> <identifier>
10.   <procedure-identifier> --> <identifier>
11.   <identifier> --> <letter><string>
12.   <string> --> <letter><string> |
                    <digit><string> |
                    <empty>
13.   <unsigned-integer> --> <digit><digits-string>
14.   <digits-string> --> <digit><digits-string> |
                    <empty>
15.   <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
                    9
16.   <letter> --> A | B | C | D | ... | Z
```

1. **Лістинг програми мовою C++**

   lexer.cpp

```cpp
#include "lexer.h"

void Lexer::lexer()
{
    init();
    string buff = "";
    int Const = 501;
```

```
int idn = 1001;
int flag = 0;
gets();
while (!finput->eof())
{

    buff = "";
    lexemCode = 0;
    start_Row = row;
    start_Col = col;
    switch (symbolCat)
    {
    case 0:// whitespace
        while (!finput->eof())
        {
            if (symbolCat != 0) break;
            gets();
        }
        break;
    case 1://constant
        while (!finput->eof() && symbolCat == 1)
        {
            buff += symbol;
            gets();
        }

        if (constants[buff])
        {
            lexemCode = constants[buff];
        }
        else
        {

            lexemCode = Const;
            constants[buff] = Const;
            Const++;

        }
        setStruct(lexemCode, buff);
        break;
    case 2://keyword or identifier
        while (!finput->eof() && (symbolCat == 1 || symbolCat == 2))
        {
            buff += symbol;
            gets();
        }
        if (keywords[buff])
        {
            lexemCode = keywords[buff];

        }
        else
        {
            if (identifiers[buff])
            {
                lexemCode = identifiers[buff];

            }
            else
            {
                lexemCode = idn;
                identifiers[buff] = lexemCode;
                idn++;

            }
```

```
            }
            setStruct(lexemCode, buff);
            break;
        case 3:
            buff = symbol;
            if (separators[buff])
            {
                lexemCode = separators[buff];

            }
            else
            {
                lexemCode = symbol;
                separators[buff] = symbol;
            }
            setStruct(lexemCode, buff);
            gets();
            break;
        case 5:
            flag = 0;
            gets();
            if (symbol == '*')
            {
                //gets();
                while (!finput->eof())
                {
                    gets();
                    if (symbol == '*')
                    {
                        //gets();
                        while (symbol == '*')
                        {
                            gets();
                            if (symbol == ')')
                            {
                                flag = 1;
                                break;
                            }
                        }
                    }
                    if (flag == 1)
                    {
                        break;
                    }
                    if (finput->eof())
                    {
                        //error
                        *foutput << "Lexer:Error(row " << start_Row << " position " <<
start_Col << ")" << " comment not closed" << endl;
                        fileOut();
                        exit(0);
                    }
                }
            }
            else
            {
                col--;
                buff = '(';
                err(buff);
                col++;

            }
            gets();
            break;
        case 6:
```

```
            //error
            buff = symbol;
            err(buff);
            gets();
            break;

        }
    }
    //fileOut();
}

void Lexer::init()
{
    for (int i = 0; i < 128; i++)// illegal
    {
        symbolCategory[i] = 6;
    }

    for (int i = 8; i <= 13; i++) // white
    {
        symbolCategory[i] = 0;
    }

    for (int i = 48; i <= 57; i++) // digit
    {
        symbolCategory[i] = 1;
    }

    for (int i = 65; i <= 90; i++) // let
    {
        symbolCategory[i] = 2;
    }
    symbolCategory[32] = 0;
    symbolCategory['('] = 5;
    symbolCategory[';'] = 3;
    symbolCategory['.'] = 3;
    symbolCategory[','] = 3;
    symbolCategory[':'] = 3;
}

void Lexer::gets()
{
    symbol = finput->get();
    symbolCat = symbolCategory[symbol];
    if (symbol == '\n')
    {
        col = 0;
        row++;
    }
    else
    {
        col++;
    }
}

void Lexer::setStruct(int code, string name)
{
    token.code = code;
    token.name = name;
    token.colm = start_Col;
    token.row = start_Row;
    lexems.push_back(token);
}

void Lexer::fileOut()
```

```
{
    for (auto& i : lexems)
    {
        *foutput << i.row << "\t" << i.colm << "\t" << i.code << "\t" << i.name << endl;
    }
}

void Lexer::err(string lexem)
{
    *foutput << "Lexer::Error(row " << row << " position " << col << ")" << " illegal
symbol: " << lexem << endl;
}

vector<Lexer::Token> Lexer::getLexems()
{
    return lexems;
}

Lexer::Lexer(ifstream* fInput, ofstream* fOutput)
{
    this->finput = fInput;
    this->foutput = fOutput;
    lexer();
}
```

lexer.h

```
#pragma once
#include <iostream>
#include <stdlib.h>
#include <fstream>
#include <string>
#include <map>
#include <vector>
using namespace std;

class Lexer {
public:
       Lexer(ifstream* fInput, ofstream* fOutput);
       struct Token
       {
              string name;
              int code;
              int row;
              int colm;
       };
       Token token;
       vector<Token> getLexems();
private:
       map<string, int> identifiers;
       map<string, int> separators;
       map<string, int> constants;
       map<string, int> keywords
       { {"PROGRAM", 401},
              {"BEGIN", 402},
              {"END", 403},
              {"LABEL", 404},
              {"GOTO", 405},
              {"LINK", 406},
              {"IN", 407},
              {"OUT", 408}
       };
       int symbolCategory[128];
       vector<Token> lexems;
```

```
        int row = 1;
        int col = 0;
        int start_Row;
        int start_Col;
        char symbol;
        int lexemCode;
        int symbolCat;
        ifstream* finput;
        ofstream* foutput;
        void lexer();
        void init();
        void gets();
        void setStruct(int code, string name);
        void fileOut();
        void err(string lexem);


};
```

main.cpp

```
#include "lexer.h"
#include "syntax.h"

int main(int argc, char* argv[])
{

    if (argc != 2)
        cout << "Error" << endl;
    else
    {

        ifstream fInput(argv[1] + string("//input.sig"));
        ofstream fOutputut(argv[1] + string("//generated.txt", ios_base::out |
ios_base::trunc));
        if (!fInput.is_open())
        {
            cout << "ERROR OF OPEN FILE" << endl;
            return 0;
        }
        Lexer lex(&fInput, &fOutputut);
        Syntax synt(&fOutputut, lex.getLexems());
        fInput.close();
        fOutputut.close();

    }
    return 1;

}
```

tree.h

```
#pragma once
#include <iostream>
#include <stdlib.h>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include "lexer.h"
using namespace std;

class Tree
{
```

```
public:
	int lexemCode;
	string terminal;
	string notTerminal;
	int depth;
	vector<Tree*> children;
	Tree(string notTerminal, string terminal, int depth, int code = -1,int row=-1,int
column=-1);
	Tree* getLastChild();
	void addNotTerm(string notTerminal, int code = -1);
	void addTerm(Lexer::Token lexem);
	void output(ofstream* fout);
	int column;
	int row;


};
```

syntax.h

```
#pragma once
#include "lexer.h"
#include "tree.h"
using namespace std;

class Syntax {
public:
	Syntax(ofstream* out, vector<Lexer::Token> lexems);
private:
	ofstream* out;
	vector<Lexer::Token> lexems;
	Lexer::Token buffer;
	Tree* root;
	vector<Lexer::Token>::iterator iterator = lexems.begin();
	void scan(string expected);
	void signalProgram();
	void program(Tree* node);
	void block(Tree* node);
	void declarations(Tree* node);
	void labelDeclarations(Tree* node);
	void labelsList(Tree* node);
	void statementsList(Tree* node);
	int statement(Tree* node);
	void variableIdentifier(Tree* node);
	void procedureIdentifier(Tree* node);
	void identifier(Tree* node);
	int unsignedInteger(Tree* node);
	void error(string expected);

};
```

syntax.cpp

```
#include "syntax.h"
#include "CodeGenerator.h"

Syntax::Syntax(ofstream* out, vector<Lexer::Token> lexems) :
	out(out),
	lexems{ lexems }
{signalProgram(); }

void Syntax::scan(string expected)
{
	if (iterator != lexems.end())
	{
```

```
		buffer = *iterator++;
	}
	else
	{
		bool tmp = lexems.empty();
		if (tmp);
		{
			*out << "Syntax: Error (file is empty)" << endl;
			out->close();
			exit(0);
		}
		iterator--;
		*out << "Syntax: Error(line: " << iterator->row << " position: " <<
iterator->colm + iterator->name.length() << ") after '" << iterator->name << "' expected
'" << expected << "'" << endl;
		out->close();
		exit(0);
	}
}

void Syntax::signalProgram()
{
	root = new Tree("<signal-program>", "", 0);
	scan("PROGRAM");
	root->addNotTerm("<program>");
	program(root->getLastChild());
	//root->output(out);
	CodeGenerator a(root, out);

}

void Syntax::program(Tree* node)
{

	if (buffer.code == 401)//program
	{
		node->addTerm(buffer);
		scan("procedure idintifier");
		node->addNotTerm("<procedure-identifier>");
		procedureIdentifier(node->getLastChild());
		scan(";");
		if (buffer.code == ';')
		{
			node->addTerm(buffer);
			scan("block");
			node->addNotTerm("<block>");
			block(node->getLastChild());
			scan(".");
			if (buffer.code == '.')
			{
				node->addTerm(buffer);
			}
			else
			{
				error(".");
			}
		}
		else
		{
			error(";");
		}
	}
	else
	{
		error("PROGRAM");
```

```
	}
}

void Syntax::block(Tree* node)
{
	node->addNotTerm("<declarations>");
	declarations(node->getLastChild());
	//scan("BEGIN");
	if (buffer.code == 402)//begin
	{
		node->addTerm(buffer);
		scan("statements-list");
		node->addNotTerm("<statements-list>");
		statementsList(node->getLastChild());
		//scan("END");
		if (buffer.code == 403)//end
		{
			node->addTerm(buffer);
		}
		else
		{
			error("END");
		}
	}
	else
	{
		error("BEGIN");
	}
}

void Syntax::declarations(Tree* node)
{
	node->addNotTerm("<label-declarations>");
	labelDeclarations(node->getLastChild());
}

void Syntax::labelDeclarations(Tree* node)
{

	//scan("LABEL");
	if (buffer.code == 404)//label
	{
		node->addTerm(buffer);
		scan("unsigned integer");
		node->addNotTerm("<unsigned-integer>");
		unsignedInteger(node->getLastChild());
		scan("labels list");
		node->addNotTerm("<labels-list>");
		labelsList(node->getLastChild());
		//scan(";");
		if (buffer.code == ';')
		{
			node->addTerm(buffer);
			scan("BEGIN");
		}
		else
		{
			error(";");
		}
	}
	else
	{
		node->addNotTerm("<empty>");
	}
}
```

```
void Syntax::labelsList(Tree* node)
{

	if (buffer.code == ',')
	{
		node->addTerm(buffer);
		scan("unsigned integer");
		node->addNotTerm("<unsigned-integer>");
		unsignedInteger(node->getLastChild());
		scan("labels list");
		node->addNotTerm("<labels-list>");
		labelsList(node->getLastChild());
	}
	else
	{
		node->addNotTerm("<empty>");
	}
}

void Syntax::statementsList(Tree* node)
{
	node->addNotTerm("<statement>");
	if (statement(node->getLastChild()))
	{
		node->addNotTerm("<statements-list>");
		statementsList(node->getLastChild());
	}
	else
	{
		node->children.pop_back();
		node->addNotTerm("<empty>");
	}
}

int Syntax::statement(Tree* node)
{

	if (buffer.code > 500 && buffer.code < 1001)
	{
		node->addNotTerm("<unsigned-integer>");
		unsignedInteger(node->getLastChild());
		scan(":");
		if (buffer.code == ':')
		{
			node->addTerm(buffer);
			scan("statment or END");
			node->addNotTerm("<statement>");
			statement(node->getLastChild());
			return 1;

		}
		else
		{
			error(":");
		}

	}
	else
	{
		switch (buffer.code)
		{
		case 405://goto
			node->addTerm(buffer);
			scan("unsigned integer");
```

```
		node->addNotTerm("<unsigned-integer>");
		unsignedInteger(node->getLastChild());
		scan(";");
		if (buffer.code == ';')
		{
			node->addTerm(buffer);
			scan("statment or END");
			return 1;
		}
		else
		{
			error(";");
		}
		break;
	case 406://link
		node->addTerm(buffer);
		scan("variable identifier");
		node->addNotTerm("<variable-identifier>");
		variableIdentifier(node->getLastChild());
		scan(",");
		if (buffer.code == ',')
		{
			node->addTerm(buffer);
			scan("unsigned integer");
			node->addNotTerm("<unsigned-integer>");
			unsignedInteger(node->getLastChild());
			scan(";");
			if (buffer.code == ';')
			{
				node->addTerm(buffer);
				scan("statment or END");
				return 1;
			}
			else
			{
				error(";");
			}
		}
		else
		{
			error(",");
		}
		break;
	case 407://in
		node->addTerm(buffer);
		scan("unsigned-integer");
		node->addNotTerm("<unsigned-integer>");
		unsignedInteger(node->getLastChild());
		scan(";");
		if (buffer.code == ';')
		{
			node->addTerm(buffer);
			scan("statment or END");
			return 1;
		}
		else
		{
			error(";");
		}
		break;
		node->addTerm(buffer);
		scan("");
	case 408://out
		node->addTerm(buffer);
		scan("unsigned-integer");
```

```
                        node->addNotTerm("<unsigned-integer>");
                        unsignedInteger(node->getLastChild());
                        scan(";");
                        if (buffer.code == ';')
                        {
                                node->addTerm(buffer);
                                scan("statment or END");
                                return 1;
                        }
                        else
                        {
                                error(";");
                        }
                        break;
                default:
                        return 0;
                }
        }
}

void Syntax::variableIdentifier(Tree* node)
{
        node->addNotTerm("<identifier>");
        identifier(node->getLastChild());

}

void Syntax::procedureIdentifier(Tree* node)
{
        node->addNotTerm("<identifier>");
        identifier(node->getLastChild());
}

void Syntax::identifier(Tree* node)
{
        if (buffer.code > 1000)
        {
                node->addTerm(buffer);
        }
        else
        {
                error("identifier");
        }
}

int Syntax::unsignedInteger(Tree* node)
{

        if (buffer.code > 500 && buffer.code < 1001)
        {
                node->addTerm(buffer);
                return 1;
        }
        else
        {
                error("unsigned integer");
        }
}

void Syntax::error(string expected)
{
        if (expected == "PROGRAM")
        {
                *out << "Syntax: Error(line: 1 position: 1) at the beginning of the program
expected 'PROGRAM'" << endl;
```

```
	}
	else
	{
		iterator = iterator - 2;
		*out << "Syntax: Error(line: " << iterator->row << " position: " <<
iterator->colm + iterator->name.length() << ") after '" << iterator->name << "' expected
'" << expected << "'" << endl;
	}
	out->close();
	exit(0);
}
```

CodeGenerator.h

```
#pragma once
#include "tree.h"

class CodeGenerator {

private:
	Tree* root;
	ofstream* fout;
	void program(Tree* node);
	vector <int> labels;
	vector <int> labels_used;
	string procedureName;
	int label, statment_flag = 0;
	int tmp = 0;
	int GOTO_flag = 0;



	void block(Tree* node);
	void label_Declarations(Tree* node);
	void labels_List(Tree* node);
	void label_unint(Tree* node);
	void statements_list(Tree* node);
	void statement(Tree* node);
	int unsigned_Integer(Tree* node);
	int GOTO_unsinden_integer(Tree* node);

public:
	CodeGenerator(Tree* root, ofstream* fout);
};
```

CodeGenerator.cpp

```
#include "CodeGenerator.h"

CodeGenerator::CodeGenerator(Tree* root, ofstream* fout)
{
	this->root = root;
	this->fout = fout;

	*fout << "push rbp\nmov rbp, rsp\n" << endl;
	program(root->getLastChild());
	*fout << "\npop rbp\nret" << endl;
}

void CodeGenerator::program(Tree* node)
{
	for (auto it : node->children)
	{

		if (it->notTerminal == "<block>")
```

```
                {
                        block(it);

                }
        }
}

void CodeGenerator::block(Tree* node)
{
        for (auto it : node->children)
        {
                if (it->notTerminal == "<declarations>")
                {
                        label_Declarations(it->children[0]);

                }
                if (it->notTerminal == "<statements-list>")
                {
                        statements_list(it);

                }
        }
}

void CodeGenerator::label_Declarations(Tree* node)
{
        for (auto it : node->children)
        {
                if (it->notTerminal == "<unsigned-integer>")
                {
                        label_unint(it);
                        continue;
                }
                if (it->notTerminal == "<labels-list>")
                {
                        labels_List(it);
                        continue;
                }
        }
}

void CodeGenerator::labels_List(Tree* node)
{
        for (auto it : node->children)
        {
                if (it->notTerminal == "<unsigned-integer>")
                {
                        label_unint(it);
                        continue;
                }
                if (it->notTerminal == "<labels-list>")
                {
                        labels_List(it);
                        continue;
                }
        }
}

void CodeGenerator::label_unint(Tree* node)
{
        label = stoi(node->children[0]->terminal);
        for (size_t i = 0; i < labels.size(); i++)
        {
                if (label == labels[i])
                {
```

```
                    *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " <<
node->children[0]->column << "). Label '" << label << "' was declared";
                    fout->close();
                    exit(0);
                }
        }
        labels.push_back(label);
}




void CodeGenerator::statements_list(Tree* node)
{
        for (auto it : node->children)
        {
                if (it->notTerminal == "<statement>")
                {
                        statment_flag = 1;
                        statement(it);

                }
                if (it->notTerminal == "<statements-list>")
                {
                        statements_list(it);

                }
                if (it->notTerminal == "<empty>")
                {
                        if (statment_flag == 0)
                        {
                                *fout << "nop" << endl;
                        }
                }

        }
}

void CodeGenerator::statement(Tree* node)
{
        for (auto it : node->children)
        {
                if (node->children[0]->terminal == "GOTO" && GOTO_flag == 1)
                {
                        GOTO_flag = 0;
                        continue;
                }
                if (it->notTerminal == "<unsigned-integer>")
                {
                        tmp = unsigned_Integer(it);
                        *fout << tmp << ": ";
                        labels_used.push_back(tmp);
                        it = node->children[2];
                        statement(it);
                        continue;
                }
                if (it->terminal == "GOTO")
                {
                        GOTO_flag = 1;
                        it = node->children[1];
                        int buff = GOTO_unsinden_integer(it);
                        *fout << "jmp " << buff << endl;
```

```
                        continue;
                }

                if (it->terminal=="LINK")
                {
                        *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " <<
node->children[0]->column << "). " << node->children[0]->terminal << "cannot be used
without SIGNAL" << endl;
                        fout->close();
                        exit(0);

                }
                if (it->terminal == "IN")
                {
                        *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " <<
node->children[0]->column << ").  " << node->children[0]->terminal << "cannot be used
without LINK " << endl;
                        fout->close();
                        exit(0);

                }
                if (it->terminal == "OUT")
                {
                        *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " <<
node->children[0]->column << ")." << node->children[0]->terminal << "cannot be used
without LINK" << endl;
                        fout->close();
                        exit(0);

                }

        }
}

int CodeGenerator::unsigned_Integer(Tree* node)
{
        for (auto it : labels)
        {
                if (it == stoi(node->children[0]->terminal))
                {
                        return it;
                }
        }

        *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " << node-
>children[0]->column << "). Label " << node->children[0]->terminal << " was not
declareted";
        fout->close();
        exit(0);
}

int CodeGenerator::GOTO_unsinden_integer(Tree* node)
{
        for (auto it : labels)
        {
                if (it == stoi(node->children[0]->terminal))
                {
                        for (auto it2 : labels_used)
                        {
                                if (it == it2)
                                {
                                        return it;
                                }
                        }
```

```
                    *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " <<
node->children[0]->column << "). Label " << node->children[0]->terminal << " was not
used";
                    fout->close();
                    exit(0);
                }
        }

        *fout << "ERROR: (row: " << node->children[0]->row << ", colm: " << node-
>children[0]->column << "). Label " << node->children[0]->terminal << " was not
declareted";
        fout->close();
        exit(0);
}
```

Тестування програми

Тест №1(без помилок)

Input.sig:

PROGRAM MYPROGRAM;
LABEL 10,30;
BEGIN
10:
30:GOTO 10;
GOTO 30;
END.

generated.txt:

push rbp

mov rbp, rsp

10: 30: jmp 10

jmp 30

pop rbp

ret

Тест №2

Input.sig:
PROGRAM MYPROGRAM;
LABEL 10,10,30;
BEGIN
END.
generated.txt:

push rbp

mov rbp, rsp

ERROR: (row: 2, colm: 10). Label '10' was declared

Тест №3

Input.sig:
PROGRAM MYPROGRAM;
LABEL 10,20,30;
BEGIN
GOTO 15;
END.
generated.txt:

push rbp
mov rbp, rsp

ERROR: (row: 4, colm: 6). Label 15 was not declareted

Тест №4

Input.sig:
PROGRAM MYPROGRAM;
LABEL 10,20,30;
BEGIN
GOTO 20;
END.
generated.txt:

push rbp
mov rbp, rsp

ERROR: (row: 4, colm: 6). Label 20 was not used
Тест №5

Input.sig:
PROGRAM MYPROGRAM;
LABEL 10,30;
BEGIN
GOTO 20;
END.
generated.txt:

push rbp
mov rbp, rsp

ERROR: (row: 4, colm: 6). Label 20 was not declareted