НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ
Кафедра системного програмування та спеціалізованих комп'ютерних систем

Лабораторна робота №2

з дисципліни
«Основи проектування трансляторів»
Тема: **«Розробка генератора коду»**

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

Київ 2021

Варіант №2

Граматика

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

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

<block>. |

PROCEDURE <procedure-identifier><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> --><variable-identifier><identifiers-
list>:<attribute><attributes-list> ;

8. <identifiers-list> --> , <variable-identifier>

<identifiers-list> |

<empty>

9. <attributes-list> --> <attribute> <attributes-list> | <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
```

Лістинг програми

Main.cpp

```
#include "Table.h"
#include "LexicalAnalyzer.h"
#include "SyntacticalAnalyzer.h"
#include "CodeGenerator.h"

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

using namespace std;

void doFile(const string& fname)
{
   cout << fname << endl;

   string filename = fname;
   string directory = filename;

   filename.append("/input.sig");
   directory.append("/generated.txt");

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

   if (f.is_open())
   {
      Table table;
      LexicalAnalyzer lexicalAnalyzer(out, f, table);
      if (lexicalAnalyzer.start())
      {
         // lexicalAnalyzer.printNodes();
         cout << "LexicalAnalyzer has been passed" << endl;
         SyntacticalAnalyzer syntacticalAnalyzer(out,
lexicalAnalyzer.lexemes, table);
         if (syntacticalAnalyzer.startSyntacticalAnalyzer())
         {
            // syntacticalAnalyzer.printNodes();
            cout << "SyntacticalAnalyzer has been passed" << endl;
            CodeGenerator codeGenerator(out, syntacticalAnalyzer.rootNode,
table);
            if (codeGenerator.startCode())
            {
               codeGenerator.printCode();
               cout << "CodeGenerator has been passed" << endl;
            }
            else
            {
               cout << "CodeGenerator has been failed" << endl;
            }
         }
         else cout << "SyntacticalAnalyzer has been failed" << endl;
      }
      else
      {
         cout << "LexicalAnalyzer has been failed" << endl;
      }
   }
   else
```

```
    {
       cout << "Unable to open file " << filename;
    }

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

// #define RUN_ALL_TESTS

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

#ifdef RUN_ALL_TESTS
    for (int i = 1; i <= 8; i++)
    {
       string filename = argv[1];
       if (i > 9)
          filename.append("/test");
       else
          filename.append("/test0");
       filename.append(to_string(i));
       doFile(filename);
    }
#else
    doFile((const char*)argv[1]);
#endif

    return 0;
}
```

CodeGenerator.h

```
//
// Created by oleks on 26.05.2021.
//

#ifndef LAB1_CODEGENERATOR_H
#define LAB1_CODEGENERATOR_H


#include <utility>
#include <stack>

#include "Table.h"
#include "SyntacticalAnalyzer.h"
#include "LexicalAnalyzer.h"

class CodeGenerator
{
private:
    static const int BODY_INDENT = 2;
    static const int DECL_INDENT = 0;

    const Node& root;
    const Table table;
    ostream& output;
```

```
   string code;
   string data;

   int indent = DECL_INDENT;

   struct variable
   {
      string name;
      PropertyLocation lex;
      vector<string> attributes;
      string base_type;

      variable(string name, PropertyLocation lex) : name(std::move(name)),
lex(lex)
      {}
   };

   vector<variable> variables;

   void printError(const Node& node, string format, ...);

   string makeCode();

   void addLine(const string& line, int newIndent = -1);

   bool collectVariables();
   bool makeMainProcedure();

   string getProgramName();

public:
   CodeGenerator(ostream& stream, const Node& root, const Table& tab) :
table(tab), output(stream), root(root)
   {}

   bool startCode();
   void printCode() const;
};


#endif //LAB1_CODEGENERATOR_H
```

CodeGenerator.cpp

```
//
// Created by oleks on 26.05.2021.
//

#include "CodeGenerator.h"

#include <cstdarg>
#include <algorithm>

void CodeGenerator::addLine(const string& line, int newIndent)
{
   if (newIndent != -1) indent = newIndent;
   for (int i = 0; i < indent; i++) code += " ";
   code += line + "\n";
```

```
}

string CodeGenerator::makeCode()
{
   return " .486\n"
          ".model flat, stdcall\n"
          ".code\n"
          "\n"
          + code +
          "\n"
          "END\n";
}

void CodeGenerator::printError(const Node& node, string format, ...)
{
   char buff[300];

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

   Node terminal = node.searchAnyTerminal();
   if (terminal.isEmpty())
      output << "Code generator error: ";
   else
      output << "Code generator error at line " << terminal.lexeme.line << ",
column " << terminal.lexeme.column << ": ";

   output << (const char*) buff << endl;
}

bool in_str_vector(const vector<string>& v, const string& str)
{
   return find(v.begin(), v.end(), str) != v.end();
}

bool CodeGenerator::collectVariables()
{
   bool has_errors = false;
   Node list = root.search("parameters_list");
   if (list.isEmpty()) return has_errors;

   vector<Node> declarations = root
         .search("declarations_list")
         .extractList("declaration", "declarations_list");

   for (const auto& decl : declarations)
   {
      vector<Node> attributes = decl.extractList("attribute",
"attributes_list");
      string base_attribute;
      vector<string> add_attributes;
      for (const auto& var : attributes)
      {
         string name = var.getKw(table);
         if (name == "INTEGER" || name == "FLOAT" || name == "BLOCKFLOAT")
         {
            if (!base_attribute.empty())
            {
               has_errors = true;
               printError(decl, "Specified %s two or more base attributes
specified at the same time", name.c_str());
            }
            else
            {
               base_attribute = name;
            }
         }
         else if (name == "EXT" || name == "SIGNAL" || name == "COMPLEX")
         {
            if (in_str_vector(add_attributes, name))
            {
               has_errors = true;
               printError(var, "Base attribute %s already declared",
name.c_str());
            }
            else
            {
               add_attributes.push_back(name);
            }
         }
         else
         {
            has_errors = true;
            printError(decl, "Unknown attribute %s", name.c_str());
            continue;
         }
      }

      vector<Node> vars = decl.extractList("variable_identifier",
"identifiers_list");
      for (const auto& var : vars)
      {
         string variable_name = var.getId(table);
         bool found = false;
         for(const auto& j : variables)
            if(j.name == variable_name) found = true;

         if(found)
         {
            has_errors = true;
            printError(var, "Variable with name %s was already declared",
variable_name.c_str());
            continue;
         }

         variable ve = variable(variable_name, var.lexeme);
         ve.base_type = base_attribute;
         ve.attributes = add_attributes;
         variables.emplace_back(ve);
      }
   }

   return has_errors;
}

string CodeGenerator::getProgramName()
{
   return root.search("procedure_identifier").getId(table);
}


bool CodeGenerator::makeMainProcedure()
{
   bool has_errors = false;
   string name = getProgramName();
   addLine("PUBLIC " + name, DECL_INDENT);
   addLine("");
   addLine(";; Main procedure declaration");

   string parameters;
   for(int i = 0; i < variables.size(); i++)
   {
      parameters += variables[i].name + ": ";
      if(variables[i].base_type == "FLOAT")
         parameters += "FLOAT";
      else if(variables[i].base_type == "BLOCKFLOAT")
         parameters += "BLOCKFLOAT";
      else if(variables[i].base_type == "INTEGER")
         parameters += "DWORD";
      else
      {
         has_errors = true;
         printError(Node(), "Unknown attribute %s", name.c_str());
         continue;
      }
      if(i != variables.size() -1)
         parameters += ", ";
   }

   addLine(name + " PROC NEAR " + parameters, DECL_INDENT);
   addLine("PUSH BP", BODY_INDENT);
   addLine("MOV BP, SP", BODY_INDENT);
   addLine("POP BP", BODY_INDENT);
   addLine("RET", BODY_INDENT);
   addLine(name + " ENDP", DECL_INDENT);

   return has_errors;
}

bool CodeGenerator::startCode()
{
   bool has_errors = false;

   has_errors = collectVariables();
   has_errors = has_errors || makeMainProcedure();

   code = makeCode();
   return !has_errors;
}

void CodeGenerator::printCode() const
{
   output << code;
}
```

Node.h

```
//
// Created by oleks on 30.04.2021.
//

#ifndef LAB1_NODE_H
#define LAB1_NODE_H

#include <vector>
#include <iterator>
#include "PropertyLocation.h"
```

```
class Node
{
private:
   static const int OFFSET = 2;

   bool empty;
   void print(const Table& tab, ostream& stream, int offset);
   static Node search(const Node& node, const string& name, bool terminal,
bool fail);
   static void extractList(const Node& node, const string& elementName, const
string& listName, vector<Node>& v);
public:
   Node() : Name("<empty>"), terminal(false), empty(true)
   {};
   Node(const string& name, const PropertyLocation& position);

   string Name;
   PropertyLocation lexeme;
   vector<Node> nodes;
   bool terminal;

   bool isEmpty() const;
   void print(const Table& tab, ostream& stream);
   void markEmpty();

   string getId(const Table& tab) const;
   string getKw(const Table& tab) const;
   int getConst(const Table& tab) const;

   Node search(const string& name) const;
   Node searchAnyTerminal() const;
   vector<Node> extractList(const string& elementName, const string&
listName) const;
};


#endif //LAB1_NODE_H
```

Node.cpp

```
//
// Created by oleks on 30.04.2021.
//

#include "Node.h"

bool Node::isEmpty() const
{
   return empty;
}

Node::Node(const string& name, const PropertyLocation& position)
{
   empty = false;
   for (auto& c: name) Name += (char) (c);
   lexeme = position;
   nodes = vector<Node>();
   terminal = false;
}
```

```
void Node::print(const Table& tab, ostream& stream)
{
   print(tab, stream, 0);
}

void Node::print(const Table& tab, ostream& stream, int offset)
{
   for (int i = 0; i < offset; i++)
      stream << '.';
   if (empty)
   {
      stream << "<" << Name << ">" << endl;
      for (int i = 0; i < offset + OFFSET; i++)
         stream << '.';
      stream << "<empty>" << endl;
   }
   else
   {
      if (terminal)
      {
         string data;
         switch (tab.classifyIndex(lexeme.id))
         {
            case IdType::DM:
               data = to_string(lexeme.id) + " " + string(1, (char)
lexeme.id);
               break;
            case IdType::Keyword:
               data = to_string(lexeme.id) + " " + tab.getKeyword(lexeme.id);
               break;
            case IdType::Id:
               data = to_string(lexeme.id) + " " + tab.getId(lexeme.id);
               break;
            default:
               data = "INVALID";
         }
         stream << data << endl;
      }
      else stream << "<" << Name << ">" << endl;
   }
   for (auto node : nodes)
   {
      node.print(tab, stream, offset + OFFSET);
   }
}

void Node::markEmpty()
{
   empty = true;
}

Node Node::search(const Node& node, const string& name, bool terminal, bool
fail)
{
   if (!fail)
   {
      if (!terminal && node.Name == name)
         return node;
      if (terminal && node.terminal)
         return node;
   }

   for (const auto& i : node.nodes)
   {
      Node r = search(i, name, terminal, false);
      if (!r.isEmpty()) return r;
   }
   return Node();
}

Node Node::search(const string& name) const
{
   if (empty) return Node();
   return search(*this, name, false, false);
}

Node Node::searchAnyTerminal() const
{
   if (empty) return Node();
   return search(*this, string(), true, false);
}

void Node::extractList(const Node& node, const string& elementName, const
string& listName, vector<Node>& v)
{
   const Node elem = search(node, elementName, false, false);
   const Node tail = search(node, listName, false, true);
   if (!elem.isEmpty()) v.push_back(elem);
   if (!tail.isEmpty()) extractList(tail, elementName, listName, v);
}

vector<Node> Node::extractList(const string& elementName, const string&
listName) const
{
   vector<Node> res;
   extractList(*this, elementName, listName, res);
   return res;
}

string Node::getId(const Table& tab) const
{
   return tab.getId(this->search("identifier").lexeme.id);
}

string Node::getKw(const Table& tab) const
{
   return tab.getKeyword(this->search("keywords").lexeme.id);
}
```

SyntacticalAnalyzer.h

```
//
// Created by oleks on 27.04.2021.
//

#ifndef LAB1_SYNTACTICALANALYZER_H
#define LAB1_SYNTACTICALANALYZER_H

#include <vector>
#include <iterator>
#include <cstdarg>
#include <algorithm>
#include <iostream>
#include "PropertyLocation.h"
#include "Node.h"
```

```
using namespace std;

class SyntacticalAnalyzer {
private:
    struct read_res {
        bool ok;
        Node data;
        string error;
    };

    const vector<PropertyLocation> lexemes;
    const Table table;
    ostream &output;
    int lex_counter;

    string createError(string format, ...);
    void appendTerminal(Node &n, const string &name, PropertyLocation lex);
    read_res readKeyword(Node &n, const string &keyword);
    read_res readDm(Node &n, char delimiter);
    read_res readIdentifier(Node &n);
    void reset();

#define DECL(name) read_res name##_func();
    DECL(root)
    DECL(signal_program)
    DECL(program)
    DECL(block)
    DECL(statement_list)
    DECL(parametrs_list)
    DECL(declarations_list)
    DECL(declaration)
    DECL(identifiers_list)
    DECL(attributes_list)
    DECL(attribute)
    DECL(procedure_identifier)
    DECL(variable_identifier)
    DECL(identifier)
#undef DECL

public:
    Node rootNode;

    SyntacticalAnalyzer(ostream &output, const vector<PropertyLocation>
&lexemes, const Table &table) : output(output),

lexemes(lexemes),

table(table) {
        reset();
    }

    bool startSyntacticalAnalyzer();
    void printNodes();
};

#endif //LAB1_SYNTACTICALANALYZER_H
```

CharType.h

```
//
// 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
```

SyntacticalAnalyzer.cpp

```
//
// Created by oleks on 27.04.2021.
//

#include "SyntacticalAnalyzer.h"

string SyntacticalAnalyzer::createError(string format, ...) {
    char buff[300];

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

    PropertyLocation p = lexemes[lex_counter];
    string result =
            "SyntacticalAnalyzer: Error ( line: " + to_string(p.line) + ",
column " + to_string(p.column) + "): ";
    result += (const char *) buff;
    return result;
}

void SyntacticalAnalyzer::appendTerminal(Node &n, const string &name,
PropertyLocation lex) {
    Node node(name, lex);
    node.terminal = true;
    n.nodes.push_back(node);
}

SyntacticalAnalyzer::read_res SyntacticalAnalyzer::readKeyword(Node &n, const
string &keyword) {
    IdType type = table.classifyIndex(lexemes[lex_counter].id);
    if (type != IdType::Keyword) {
        string error = createError("Keyword lexeme expected, but %s found",
idTypeToString(type).c_str());
        read_res readRes;
        readRes.ok = false;
        readRes.error = error;
        return readRes;
    }
    string key;
    if ((key = table.getKeyword(lexemes[lex_counter].id)) != keyword) {
        string error = createError("'%s' keyword expected, but %s keyword
found", keyword.c_str(), key.c_str());
        read_res readRes;
        readRes.ok = false;
        readRes.error = error;
        return readRes;
    }
    appendTerminal(n, "keywords", lexemes[lex_counter++]);
    return {
            .ok = true,
    };
}

SyntacticalAnalyzer::read_res SyntacticalAnalyzer::readDm(Node &n, const char
delimiter) {
    IdType type = table.classifyIndex(lexemes[lex_counter].id);
    if (type != IdType::DM) {
        string error = createError("Delimiter lexeme expected, but %s found",
idTypeToString(type).c_str());
        read_res readRes;
        readRes.ok = false;
        readRes.error = error;
        return readRes;
    }
    if (lexemes[lex_counter].id != delimiter) {
        string error = createError("'%c' keyword expected, but %c keyword
found", (char) delimiter,
                                   (char) lexemes[lex_counter].id);
        read_res readRes;
        readRes.ok = false;
        readRes.error = error;
        return readRes;
    }
    appendTerminal(n, "delimiter", lexemes[lex_counter++]);
    return {
            .ok = true,
    };
}


SyntacticalAnalyzer::read_res SyntacticalAnalyzer::readIdentifier(Node &n) {
    IdType type = table.classifyIndex(lexemes[lex_counter].id);
    if (type != IdType::Id) {
        string error = createError("Identifier lexeme expected, but %s
found", idTypeToString(type).c_str());
        read_res readRes;
        readRes.ok = false;
        readRes.error = error;
        return readRes;
    }
    appendTerminal(n, "identifier", lexemes[lex_counter++]);
    return {
            .ok = true,
    };
}


#define DECL(name) SyntacticalAnalyzer::read_res
SyntacticalAnalyzer::name##_func() { Node node(#name, lexemes[lex_counter]);
read_res rr; int old = lex_counter;
#define ENDDECL return { .ok = true, .data = node }; }
#define READ(expr) if(!(rr = (expr)).ok) { return rr; }
#define READP(expr) READ(expr) else node.nodes.push_back(rr.data);
#define FALLBACK { node.nodes.clear(); lex_counter = old; }

DECL(root)
    {
        return signal_program_func();
    }
ENDDECL

DECL(signal_program)
    {
        READP(program_func());
    }
ENDDECL

DECL(program)
    {
        if (!readKeyword(node, "PROGRAM").ok) {
            FALLBACK;
            READ(readKeyword(node, "PROCEDURE"));
            READP(procedure_identifier_func());
            READP(parametrs_list_func());
            READ(readDm(node, ';'));
            READP(block_func());
            READ(readDm(node, ';'));
        } else {
            READP(procedure_identifier_func());
            READ(readDm(node, ';'));
            READP(block_func());
            READ(readDm(node, '.'));
        }

    }
ENDDECL

DECL(block)
    {
        READ(readKeyword(node, "BEGIN"));
        READP(statement_list_func());
        READ(readKeyword(node, "END"));
    }
ENDDECL

DECL(statement_list)
    {
        FALLBACK;
        node.markEmpty();
    }
ENDDECL

DECL(parametrs_list)
    {
        if (readDm(node, '(').ok) {
            READP(declarations_list_func());
            READ(readDm(node, ')'));
        } else {
            FALLBACK;
            node.markEmpty();
        }
    }
ENDDECL

DECL(declarations_list)
    {
        if ((rr = declaration_func()).ok) {
            node.nodes.push_back(rr.data);
            READP(declarations_list_func());
        } else {
            FALLBACK;
            node.markEmpty();
        }
    }
ENDDECL

DECL(declaration)
    {
        READP(variable_identifier_func());
        READP(identifiers_list_func());
        READ(readDm(node, ':'));
        READP(attribute_func());
        READP(attributes_list_func());
        READ(readDm(node, ';'))
    }
ENDDECL

DECL(identifiers_list)
    {
        if (readDm(node, ',').ok) {
            READP(variable_identifier_func());
            READP(identifiers_list_func());
        } else {
            FALLBACK;
            node.markEmpty();
        }
    }
ENDDECL

DECL(attributes_list)
    {
        if ((rr = attribute_func()).ok) {
            node.nodes.push_back(rr.data);
            READP(attributes_list_func());
        } else {
            FALLBACK;
            node.markEmpty();
        }
    }
ENDDECL


DECL(attribute)
    {
        if (!readKeyword(node, "SIGNAL").ok) {
            FALLBACK;
            if (!readKeyword(node, "COMPLEX").ok) {
                FALLBACK;
                if (!readKeyword(node, "INTEGER").ok) {
                    FALLBACK;
                    if (!readKeyword(node, "BLOCKFLOAT").ok) {
                        FALLBACK;
                        if (!readKeyword(node, "FLOAT").ok) {
                            FALLBACK;
                            READ(readKeyword(node, "EXT"));
                        }
                    }
                }
            }
        }
    }
ENDDECL


DECL(variable_identifier)
    {
        READP(identifier_func());
    }
ENDDECL

DECL(procedure_identifier)
    {
        READP(identifier_func());
    }
ENDDECL

DECL(identifier)
    {
        READ(readIdentifier(node));
    }
ENDDECL


#undef DECL
#undef ENDDECL

bool SyntacticalAnalyzer::startSyntacticalAnalyzer() {
    read_res rr = root_func();
    if (!rr.ok) {
        output << rr.error;
    } else {
        rootNode = rr.data;
    }
    return rr.ok;
}

void SyntacticalAnalyzer::printNodes() {
    rootNode.print(table, output);
}

void SyntacticalAnalyzer::reset() {
    lex_counter = 0;
}
```

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

```
//
// 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

```
//
// 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

```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;
}
```

Контрольні приклади

Test01

input.sig

```
PROGRAM SIG01;

BEGIN (* AS *)

END.
```

Згенерований код

```
.486
.model flat, stdcall
.code

PUBLIC SIG01

;; Main procedure declaration
SIG01 PROC NEAR
  PUSH BP
  MOV BP, SP
  POP BP
  RET
SIG01 ENDP

ENB
```

Test02

input.sig

```
PROCEDURE SIG01 ( VAR01 : FLOAT; );
BEGIN (* AS *)
END;
```

Згенерований код

```
.486
.model flat, stdcall
.code

PUBLIC SIG01

;; Main procedure declaration
SIG01 PROC NEAR VAR01: FLOAT
  PUSH BP
  MOV BP, SP
  POP BP
  RET
SIG01 ENDP

END
```

Test03

input.sig

```
PROCEDURE SIG01 ( VAR01, VAR02 : FLOAT;);
BEGIN (* AS *)
END;
```

Згенерований код

```
.486
.model flat, stdcall
.code

PUBLIC SIG01

;; Main procedure declaration
SIG01 PROC NEAR VAR01: FLOAT, VAR02: FLOAT
  PUSH BP
  MOV BP, SP
  POP BP
  RET
SIG01 ENDP

END
```

Test04

input.sig

```
PROCEDURE SIG01 ( );
BEGIN (* AS *)
END;
```

Згенерований код

```
.486
.model flat, stdcall
.code

PUBLIC SIG01

;; Main procedure declaration
SIG01 PROC NEAR
  PUSH BP
  MOV BP, SP
  POP BP
  RET
SIG01 ENDP

END
```

Test05

input.sig

```
PROCEDURE SIG01 ( VAR01, VAR02, VAR03, VAR04 : FLOAT COMPLEX INTEGER EXT;);
BEGIN (* AS *)
END;
```

Згенерований код

```
Code generator error at line 1, column 19: Specified INTEGER two or more base attributes specified at the same time
```

Test06

input.sig

```
PROCEDURE SIG01 ( VAR01, VAR02, VAR03, VAR04 : FLOAT COMPLEX EXT;
                   VAR05, VAR06 : INTEGER;
                         VAR07, VAR08 : SIGNAL BLOCKFLOAT;);
BEGIN (* AS *)
END;
```

Згенерований код

```
.486
.model flat, stdcall
.code

PUBLIC SIG01

;; Main procedure declaration
SIG01 PROC NEAR VAR01: FLOAT, VAR02: FLOAT, VAR03: FLOAT, VAR04: FLOAT, VAR05: DWORD, VAR06: DWORD, VAR07: BLOCKFLOAT, VAR08: BLOCKFLOAT
  PUSH BP
  MOV BP, SP
  POP BP
  RET
SIG01 ENDP

END
```

Test07

input.sig

```
PROCEDURE SIG01 ( VAR01, VAR02, VAR03, VAR04 : FLOAT COMPLEX EXT;

                  VAR05, VAR07 : INTEGER;

                        VAR07, VAR08 : SIGNAL BLOCKFLOAT;);

BEGIN (* AS *)

END;
```

Згенерований код

```
Code generator error at line 3, column 5: Variable with name VAR07 was already
declared
```

Test08

input.sig

```
PROCEDURE SIG01 ( VAR01, VAR02, VAR03, VAR04 : FLOAT FLOAT COMPLEX COMPLEX
EXT;

                  VAR05, VAR07 : INTEGER;

                        VAR07, VAR08 : SIGNAL BLOCKFLOAT;);

BEGIN (* AS *)

END;
```

Згенерований код

```
Code generator error at line 1, column 54: Specified FLOAT two or more base
attributes specified at the same time

Code generator error at line 1, column 68: Base attribute COMPLEX already declared

Code generator error at line 3, column 5: Variable with name VAR07 was already
declared
```