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

**Кафедра системного програмування та спеціалізованих комп’ютерних систем**

# Розрахункова-графічна робота

з дисципліни

**«Основи проектування трансляторів»**

Тема: **«Розробка синтаксичного аналізатора»**

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 <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
```

## Алгоритм синтаксичного розбору

2 – низхідний розбір за алгоритмом рекурсивного спуску.

# Лістинг програми

## Main.cpp

```
#include "Table.h"
#include "LexicalAnalyzer.h"
#include "SyntacticalAnalyzer.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);
    }
        cout << filename << endl;
        string directory = filename;
        //string directoryLex = filename;

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

        ifstream f(filename);
        ofstream out(directory);
        //ofstream outLex(directoryLex);

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

        f.close();
        out.close();
        //outLex.close();
    return 0;
}
```

# 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

```
//
// 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.
```

## Рядок лексем

```
  1 |   1 |     400 | PROGRAM
  1 |   9 |    1000 | SIG01
  1 |  14 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      46 | .
```

## Дерево розбору

```
<signal_program>
..<program>
....400 PROGRAM
....<procedure_identifier>
......<identifier>
........1000 SIG01
....59 ;
....<block>
......402 BEGIN
......<statement_list>
........<empty>
......403 END
....46 .
```

# Test02

# input.sig

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

# Рядок лексем

```
  1 |   1 |     401 | PROCEDURE
  1 |  11 |    1000 | SIG01
  1 |  18 |      40 | (
  1 |  19 |    1001 | VAR01
  1 |  25 |      58 | :
  1 |  27 |     407 | FLOAT
  1 |  32 |      59 | ;
  1 |  34 |      41 | )
  1 |  35 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      59 | ;
```

# Дерево розбору

```
<signal_program>
..<program>
....401 PROCEDURE
....<procedure_identifier>
......<identifier>
........1000 SIG01
....<parametrs_list>
......40 (
......<declarations_list>
........<declaration>
..........<variable_identifier>
............<identifier>
..............1001 VAR01
..........<identifiers_list>
............<empty>
..........58 :
..........<attribute>
............407 FLOAT
..........<attributes_list>
............<empty>
..........59 ;
........<declarations_list>
..........<empty>
......41 )
....59 ;
....<block>
......402 BEGIN
......<statement_list>
........<empty>
......403 END
....59 ;
```

## Test03

## input.sig

```
PROCEDURE SIG01
BEGIN
END.
```

## Рядок лексем

```
    1 |   1 |     401 | PROCEDURE
  1 |  11 |    1000 | SIG01
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      46 | .
```

# Дерево розбору

```
SyntacticalAnalyzer: Error ( line: 2, column 1): Delimiter lexeme expected, but
keyword found
```

# Test04

# input.sig

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

# Рядок лексем

```
  1 |   1 |     401 | PROCEDURE
  1 |  11 |    1000 | SIG01
  1 |  18 |      40 | (
  1 |  19 |    1001 | VAR01
  1 |  24 |      44 | ,
  1 |  26 |    1002 | VAR02
  1 |  32 |      58 | :
  1 |  34 |     407 | FLOAT
  1 |  39 |      59 | ;
  1 |  40 |      41 | )
  1 |  41 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      59 | ;
```

# Дерево розбору

```
<signal_program>
..<program>
....401 PROCEDURE
....<procedure_identifier>
......<identifier>
........1000 SIG01
....<parametrs_list>
......40 (
......<declarations_list>
........<declaration>
..........<variable_identifier>
............<identifier>
..............1001 VAR01
..........<identifiers_list>
............44 ,
............<variable_identifier>
..............<identifier>
................1002 VAR02
............<identifiers_list>
..............<empty>
..........58 :
..........<attribute>
............407 FLOAT
..........<attributes_list>
............<empty>
..........59 ;
........<declarations_list>
..........<empty>
......41 )
....59 ;
....<block>
......402 BEGIN
......<statement_list>
........<empty>
......403 END
....59 ;
```

# Test05

# input.sig

```
PROGRAM SIG01;

BEGIN (* AS *)

END:
```

# Рядок лексем

```
  1 |   1 |     400 | PROGRAM
  1 |   9 |    1000 | SIG01
  1 |  14 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      58 | :
```

# Дерево розбору

```
SyntacticalAnalyzer: Error ( line: 3, column 4): '.' delimiter expected, but :
delimiter found
```

# Test06

# input.sig

```
PROGRAM;

BEGIN (* AS *)

END;
```

# Рядок лексем

```
  1 |   1 |     400 | PROGRAM
  1 |   8 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      59 | ;
```

# Дерево розбору

```
SyntacticalAnalyzer: Error ( line: 1, column 8): Identifier lexeme expected, but
delimiter found
```

# Test07

# input.sig

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

# Рядок лексем

```
  1 |   1 |     401 | PROCEDURE
  1 |  11 |    1000 | SIG01
  1 |  18 |      40 | (
  1 |  19 |      41 | )
  1 |  20 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      59 | ;
```

# Дерево розбору

```
<signal_program>
..<program>
....401 PROCEDURE
....<procedure_identifier>
......<identifier>
........1000 SIG01
....<parametrs_list>
......40 (
......<declarations_list>
........<empty>
......41 )
....59 ;
....<block>
......402 BEGIN
......<statement_list>
........<empty>
......403 END
....59 ;
```

# Test08

## input.sig

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

## Рядок лексем

```
  1 |    1 |      401 | PROCEDURE
  1 |   11 |     1000 | SIG01
  1 |   18 |       40 | (
  1 |   19 |     1001 | VAR01
  1 |   24 |       44 | ,
  1 |   26 |     1002 | VAR02
  1 |   31 |       44 | ,
  1 |   33 |     1003 | VAR03
  1 |   38 |       44 | ,
  1 |   40 |     1004 | VAR04
  1 |   46 |       58 | :
  1 |   48 |      407 | FLOAT
  1 |   54 |      405 | COMPLEX
  1 |   62 |      406 | INTEGER
  1 |   70 |      409 | EXT
  1 |   73 |       59 | ;
  1 |   74 |       41 | )
  1 |   75 |       59 | ;
  2 |    1 |      402 | BEGIN
  3 |    1 |      403 | END
  3 |    4 |       59 | ;
```

# Дерево розбору

```
<signal_program>
..<program>
....401 PROCEDURE
....<procedure_identifier>
......<identifier>
........1000 SIG01
....<parametrs_list>
......40 (
......<declarations_list>
........<declaration>
..........<variable_identifier>
............<identifier>
..............1001 VAR01
..........<identifiers_list>
............44 ,
............<variable_identifier>
..............<identifier>
................1002 VAR02
............<identifiers_list>
..............44 ,
..............<variable_identifier>
................<identifier>
..................1003 VAR03
..............<identifiers_list>
................44 ,
................<variable_identifier>
..................<identifier>
....................1004 VAR04
................<identifiers_list>
..................<empty>
..........58 :
..........<attribute>
............407 FLOAT
..........<attributes_list>
............<attribute>
..............405 COMPLEX
............<attributes_list>
..............<attribute>
................406 INTEGER
..............<attributes_list>
................<attribute>
..................409 EXT
................<attributes_list>
..................<empty>
..........59 ;
........<declarations_list>
..........<empty>
......41 )
....59 ;
....<block>
......402 BEGIN
......<statement_list>
........<empty>
......403 END
....59 ;
```

# Test09

## input.sig

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

## Рядок лексем

```
  1 |   1 |      401 | PROCEDURE
  1 |  11 |     1000 | SIG01
  1 |  18 |       40 | (
  1 |  19 |     1001 | VAR01
  1 |  24 |       44 | ,
  1 |  26 |     1002 | VAR02
  1 |  31 |       44 | ,
  1 |  33 |     1003 | VAR03
  1 |  38 |       44 | ,
  1 |  40 |     1004 | VAR04
  1 |  46 |       58 | :
  1 |  48 |      407 | FLOAT
  1 |  54 |      405 | COMPLEX
  1 |  62 |      406 | INTEGER
  1 |  70 |      409 | EXT
  1 |  73 |       59 | ;
  1 |  74 |       41 | )
  1 |  75 |       59 | ;
  3 |   1 |      403 | END
  3 |   4 |       59 | ;
```

## Дерево розбору

```
SyntacticalAnalyzer: Error ( line: 3, column 1): 'BEGIN' keyword expected, but END
keyword found
```

# Test10

## input.sig

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

## Рядок лексем

```
  1 |   1 |    1000 | SIG01
  1 |   8 |      40 | (
  1 |   9 |    1001 | VAR01
  1 |  14 |      44 | ,
  1 |  16 |    1002 | VAR02
  1 |  21 |      44 | ,
  1 |  23 |    1003 | VAR03
  1 |  28 |      44 | ,
  1 |  30 |    1004 | VAR04
  1 |  36 |      58 | :
  1 |  38 |     407 | FLOAT
  1 |  44 |     405 | COMPLEX
  1 |  52 |     406 | INTEGER
  1 |  60 |     409 | EXT
  1 |  63 |      59 | ;
  1 |  64 |      41 | )
  1 |  65 |      59 | ;
  2 |   1 |     402 | BEGIN
  3 |   1 |     403 | END
  3 |   4 |      59 | ;
```

## Дерево розбору

```
SyntacticalAnalyzer: Error ( line: 1, column 1): Keyword lexeme expected, but
identifier found
```