**Варіант 21**

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

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

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

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

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

6. <declaration> --><variableidentifier>:<attribute><attributeslist> ;

7. <attributes-list> --> <attribute> <attributeslist> | <empty>

8. <attribute> --> INTEGER | FLOAT | [<range>]

9. <range> --> <unsigned-integer> .. <unsignedinteger>

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

11. <statement> --> <variable> := <expression> ; | LOOP <statements-list> ENDLOOP ;

12. <expression> --> <variable> | <unsigned-integer>

13. <variable> --> <variable-identifier><dimension>

14. <dimension> --> [ <expression> ] | <empty>

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

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

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

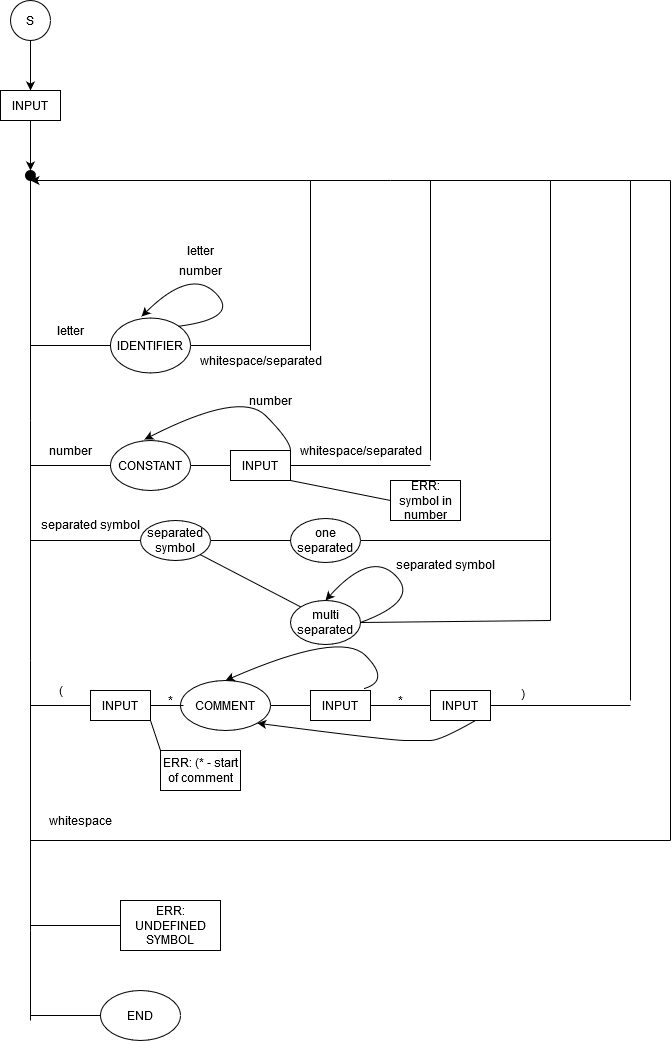
18. <string> --> <letter><string> | <digit><string> | <empty>

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



**Лістинг програми**

**OPT\_Lexer.cpp**

#include <iostream>

#include "Lexer.h"

int main() {

Lexer lexer;

lexer.scanFile("program.yandroidUA");

std::cout << std::endl << std::endl << std::endl << "\t\t\t\t" << "RESULT" << std::endl;

lexer.printScanResult();

}

**Lexer.h**

#pragma once

#include <vector>

#include <fstream>

#include <map>

#include "MultiSeparatedToken.h"

#include "LexerResult.h"

#include "TokenStatus.h"

class Lexer {

private:

std::vector<int> lettersVector = {

'a', 'A', 'b', 'B', 'c', 'C', 'd', 'D', 'e', 'E', 'f', 'F', 'g', 'G', 'h', 'H',

'k', 'K', 'l', 'L', 'm', 'M', 'n', 'N', 'o', 'O', 'p', 'P', 'q', 'Q', 'r', 'R',

's', 'S', 't', 'T', 'u', 'U', 'v', 'V', 'w', 'W', 'x', 'X', 'y', 'y', 'z', 'Z',

'i', 'I'

};

std::vector<int> numbersVector = {48, 49, 50, 51, 52, 53, 54, 55, 56, 57};

std::vector<int> whiteSpacesVector = {9, 10, 13, 32};

std::vector<int> oneSeparatedTokens = {':', ';', '[',']','.'};

std::vector<MultiSeparatedToken> multiSeparatedTokens = {

MultiSeparatedToken(':', '='),

MultiSeparatedToken('.', '.')

};

std::vector<LexerResult> lexerResultValues;

std::map<std::string, int> reservedWords;

std::map<std::string, int> multiSeparatedTokensMap;

std::map<std::string, int> oneSeparatedTokensMap;

std::map<std::string, int> identifiersTokensMap;

std::map<std::string, int> constantTokensMap;

// search ASCII code of param in lettersVector and if found return TRUE else FALSE

bool isLetter(int);

// search ASCII code of param in numbersVector and if found return TRUE else FALSE

bool isNumber(int);

// return TRUE if param is '('

bool isComment(int);

// search ASCII code of param in whiteSpacesVector and if found return TRUE else FALSE

bool isWhiteSpace(int);

// search ASCII code of param in oneSeparatedTokens and if found return TRUE else FALSE

bool isOneSeparated(int);

// search ASCII code of param in multiSeparatedTokens and if found return TRUE else FALSE

bool isMultiSeparated(int);

// search ASCII code of param in multiSeparatedTokens and if found return TRUE else FALSE

bool isSecondPartOfMultiSeparated(int, int);

// set status of token to 0

void resetTokenStatus();

// check status of token and depends on value add token as: ReservedWord, Identifier, OneSeparatedToken or MultiSeparatedToken

// invoke addTokenToResult to add token to result vector

void addToken(std::string&);

// add param of func to reservedWords

void addReservedWord(const char\*);

// add param of func to oneSeparatedTokensMap

void addOneSeparatedToken(const char\*);

// add param of func to multiSeparatedTokensMap

void addMultiSeparatedToken(const char\*);

// add token to identifiersTokensMap, returns code of just added identifier

int addIdentifier(std::string&);

// add token to constantTokensMap, returns code of just added constant

int addConstant(std::string&);

// add token to ResultsVector

void addTokenToResultVector(std::string&, TokenStatus, int, int);

// check if passed token (param of func) is contains in: \_ and return code if contains and -1 if not

// \_ = reservedWords

int isTokenReservedWord(std::string&);

// \_ = oneSeparatedTokensMap

int isTokenOneSeparatedToken(std::string&);

// \_ = multiSeparatedTokensMap

int isTokenMultiSeparatedToken(std::string&);

// \_ = identifiersTokensMap

int isTokenIdentifier(std::string&);

// \_ = constantTokensMap

int isTokenConstant(std::string&);

// return ASCII code of next character from file

int readCharacterFromFile(std::ifstream&);

// params of func is OneSeparated token

// this func check if param can be MultiSeparatedToken and if it is invoke caseMultiSeparated

// returns ASCII code of NOT letter OneSeparated, but before invoke addToken and add scanned token

int caseOneSeparated(int, std::ifstream&);

// params of func is OneSeparated token

// this func check if param can be MultiSeparatedToken and if it is invoke readCharacterFromFile

// if readed character is pair for MultiSeparatedToken add token and return nextLetter using readCharacterFromFile

// returns ASCII code of NOT MultiSeparated letter, but before invoke addToken and add scanned token

int caseMultiSeparated(int, std::ifstream&);

// params of func is letter token

// this func check if param can be ReservedWord or Ientifier

// returns ASCII code of NOT letter letter, but before invoke addToken and add scanned token

int caseLetter(int, std::ifstream&);

// params of func is letter token

// this func check next letter and if it isNumber == true add value to token, else

// returns ASCII code of NOT number letter, but before invoke addToken and add scanned token

int caseNumber(int, std::ifstream&);

// params of func is '(' letter

// check nextLetter and if it is not '\*' print ERROR

// else scan next letter untill ['\*',')'] will be scanned one by one

// returns ASCII code of letter that don't in comment block, but before invoke addToken and add scanned token

int caseComment(int, std::ifstream&);

public:

Lexer();

~Lexer() = default;

void scanFile(const char\*);

void printScanResult();

};

**Lexer.cpp**

#include "Lexer.h"

#include <string>

#include <iostream>

typedef std::pair<std::string, int> MapPair;

int TOKEN\_STATUS\_CONSTANT = 0;

int TOKEN\_STATUS\_IDENTIFIER = 1;

int TOKEN\_STATUS\_RESERVED\_WORD = 2;

int TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN = 3;

int TOKEN\_STATUS\_MULTI\_SEPARATED\_TOKEN = 4;

int TOKEN\_STATUS\_COMMENT = 5;

int oneSymbolTokenIndex = 0;

int multiSymbolTokenIndex = 301;

int reservedWordsIndex = 401;

int constantsIndex = 501;

int variablesIndex = 1001;

int currentTokenState[6] = {0, 0, 0, 0, 0, 0};

std::string token = "";

int currentColumn = -1;

int currentRow = 0;

int savedColumn = currentColumn;

int savedRow = currentRow;

bool errorHappened = false;

bool Lexer::isLetter(int character) {

for (auto& c : lettersVector) {

if (c == character) {

return true;

}

}

return false;

}

bool Lexer::isNumber(int character) {

for (auto& number : numbersVector) {

if ((char) number == character) {

return true;

}

}

return false;

}

bool Lexer::isWhiteSpace(int character) {

for (auto& whiteSpaceId : whiteSpacesVector) {

if (whiteSpaceId == character) {

return true;

}

}

return false;

}

bool Lexer::isOneSeparated(int c) {

for (auto& oneSeparatedToken : oneSeparatedTokens) {

if (oneSeparatedToken == c) {

return true;

}

}

return false;

}

bool Lexer::isMultiSeparated(int c) {

for (auto& multiSeparatedToken : multiSeparatedTokens) {

if (multiSeparatedToken.getCode() == c) {

return true;

}

}

return false;

}

bool Lexer::isSecondPartOfMultiSeparated(int sepToken, int c) {

for (auto& multiSeparatedToken : multiSeparatedTokens) {

if (multiSeparatedToken.getCode() == sepToken && multiSeparatedToken.getPairCode() == c) {

return true;

}

}

return false;

}

void Lexer::resetTokenStatus() {

currentTokenState[0] = 0;

currentTokenState[1] = 0;

currentTokenState[2] = 0;

currentTokenState[3] = 0;

currentTokenState[4] = 0;

currentTokenState[5] = 0;

}

Lexer::Lexer() {

addReservedWord("PROGRAM");

addReservedWord("BEGIN");

addReservedWord("END");

addReservedWord("VAR");

addReservedWord("FLOAT");

addReservedWord("LOOP");

addReservedWord("ENDLOOP");

addOneSeparatedToken(":");

addOneSeparatedToken(";");

addOneSeparatedToken("[");

addOneSeparatedToken("]");

addOneSeparatedToken(".");

addMultiSeparatedToken(":=");

addMultiSeparatedToken("..");

}

void Lexer::addToken(std::string&) {

if (currentTokenState[TOKEN\_STATUS\_CONSTANT] == 1) {

// std::cout << token << " is constant, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, CONSTANT, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_IDENTIFIER] == 1 && currentTokenState[TOKEN\_STATUS\_RESERVED\_WORD] == 0) {

// std::cout << token << " is identifier, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, IDENTIFIER, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_RESERVED\_WORD] == 1 && currentTokenState[TOKEN\_STATUS\_IDENTIFIER] == 0) {

// std::cout << token << " is reserved word, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, RESERVED\_WORD, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN] == 1) {

// std::cout << token << " is one separated word, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, ONE\_SEPARATED\_TOKEN, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_MULTI\_SEPARATED\_TOKEN] == 1) {

// std::cout << token << " is multi separated word, row=" << savedRow << " column=" << savedColumn << std::endl;

addTokenToResultVector(token, MULTI\_SEPARATED\_TOKEN, savedColumn, savedRow);

return;

}

if (currentTokenState[TOKEN\_STATUS\_COMMENT] == 1) {

// std::cout << token << " is comment, row=" << savedRow << " column=" << savedColumn << std::endl;

return;

}

int reservedWordCode = isTokenReservedWord(token);

if (reservedWordCode != -1) {

// std::cout << token << " is reserved word, row=" << savedRow << "column=" << savedColumn << std::endl;

addTokenToResultVector(token, RESERVED\_WORD, savedColumn, savedRow);

return;

} else {

// std::cout << token << " is identifier, row=" << savedRow << "column=" << savedColumn << std::endl;

addTokenToResultVector(token, IDENTIFIER, savedColumn, savedRow);

return;

}

}

int Lexer::addIdentifier(std::string& token) {

identifiersTokensMap.insert(MapPair(token, variablesIndex));

int code = variablesIndex;

variablesIndex++;

return code;

}

int Lexer::addConstant(std::string& word) {

constantTokensMap.insert(MapPair(word, constantsIndex));

int code = constantsIndex;

constantsIndex++;

return code;

}

void Lexer::addTokenToResultVector(std::string& token, TokenStatus status, int column, int row) {

int code = -1;

switch (status) {

case IDENTIFIER:

code = isTokenIdentifier(token);

if (code == -1) {

code = addIdentifier(token);

}

lexerResultValues.push\_back(LexerResult(token, code, row, column));

break;

case RESERVED\_WORD:

code = isTokenReservedWord(token);

lexerResultValues.push\_back(LexerResult(token, code, row, column));

break;

case ONE\_SEPARATED\_TOKEN:

code = isTokenOneSeparatedToken(token);

lexerResultValues.push\_back(LexerResult(token, code, row, column));

break;

case MULTI\_SEPARATED\_TOKEN:

code = isTokenMultiSeparatedToken(token);

lexerResultValues.push\_back(LexerResult(token, code, row, column));

break;

case CONSTANT:

code = isTokenConstant(token);

if (code == -1) {

code = addConstant(token);

}

lexerResultValues.push\_back(LexerResult(token, code, row, column));

break;

}

}

void Lexer::addReservedWord(const char\* word) {

reservedWords.insert(MapPair(word, reservedWordsIndex));

reservedWordsIndex++;

}

void Lexer::addOneSeparatedToken(const char\* word) {

oneSeparatedTokensMap.insert(MapPair(word, oneSymbolTokenIndex));

oneSymbolTokenIndex++;

}

void Lexer::addMultiSeparatedToken(const char\* word) {

multiSeparatedTokensMap.insert(MapPair(word, multiSymbolTokenIndex));

multiSymbolTokenIndex++;

}

int Lexer::isTokenReservedWord(std::string& word){

for (auto& resWord : reservedWords) {

if (resWord.first == word) {

return resWord.second;

}

}

return -1;

}

int Lexer::isTokenOneSeparatedToken(std::string& word) {

for (auto& oneSeparetadToken : oneSeparatedTokensMap) {

if (oneSeparetadToken.first == word) {

return oneSeparetadToken.second;

}

}

return -1;

}

int Lexer::isTokenMultiSeparatedToken(std::string& word) {

for (auto& multiSeparatedToken : multiSeparatedTokensMap) {

if (multiSeparatedToken.first == word) {

return multiSeparatedToken.second;

}

}

return -1;

}

int Lexer::isTokenIdentifier(std::string& word) {

for (auto& indentifier : identifiersTokensMap) {

if (indentifier.first == word) {

return indentifier.second;

}

}

return -1;

}

int Lexer::isTokenConstant(std::string& word) {

for (auto& constant : constantTokensMap) {

if (constant.first == word) {

return constant.second;

}

}

return -1;

}

int Lexer::readCharacterFromFile(std::ifstream &file) {

char currentCaracter = file.eof();

if (!file.eof()) {

file.get(currentCaracter);

currentColumn++;

}

return (int)currentCaracter;

}

int Lexer::caseMultiSeparated(int letter, std::ifstream& file) {

token += (char)letter;

while (!file.eof()) {

int nextLetter = readCharacterFromFile(file);

if (isSecondPartOfMultiSeparated(letter, nextLetter)) {

currentTokenState[TOKEN\_STATUS\_MULTI\_SEPARATED\_TOKEN] = 1;

currentTokenState[TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN] = 0;

token += (char)nextLetter;

} else {

if (!token.empty()) {

addToken(token);

}

token = "";

resetTokenStatus();

return nextLetter;

}

}

}

int Lexer::caseOneSeparated(int letter, std::ifstream& file) {

resetTokenStatus();

currentTokenState[TOKEN\_STATUS\_ONE\_SEPARATED\_TOKEN] = 1;

while (!file.eof()) {

if (isMultiSeparated(letter)) {

return caseMultiSeparated(letter, file);

} else {

token = (char)letter;

addToken(token);

token = "";

resetTokenStatus();

return readCharacterFromFile(file);

}

}

}

int Lexer::caseLetter(int letter, std::ifstream& file) {

token += (char)letter;

currentTokenState[TOKEN\_STATUS\_IDENTIFIER] = 1;

currentTokenState[TOKEN\_STATUS\_RESERVED\_WORD] = 1;

while (!file.eof()) {

int nextLetter = readCharacterFromFile(file);

if (isNumber(nextLetter)) {

currentTokenState[TOKEN\_STATUS\_IDENTIFIER] = 1;

token += (char)nextLetter;

continue;

}

if (!isLetter(nextLetter)) {

if (!token.empty()) {

addToken(token);

}

token = "";

resetTokenStatus();

return nextLetter;

}

token += (char)nextLetter;

}

}

int Lexer::caseNumber(int letter, std::ifstream& file) {

currentTokenState[TOKEN\_STATUS\_CONSTANT] = 1;

token += (char)letter;

while (!file.eof()) {

int nextLetter = readCharacterFromFile(file);

if (!isNumber(nextLetter)) {

if (!isOneSeparated(nextLetter) && !isWhiteSpace(nextLetter)) {

std::cout << "ERROR\_3 " << (char)nextLetter << " code " << nextLetter << " is not number, whitespace or separator (row:"

<< currentRow << ", col:" << currentColumn << ")" << std::endl;

errorHappened = true;

return nextLetter;

}

if (!token.empty()) {

addToken(token);

}

token = "";

resetTokenStatus();

return nextLetter;

}

token += (char)nextLetter;

}

}

int Lexer::caseComment(int letter, std::ifstream& file) {

resetTokenStatus();

currentTokenState[TOKEN\_STATUS\_COMMENT] = 1;

token += (char)letter;

bool isStartFound = false;

int nextLetter = readCharacterFromFile(file);

if (nextLetter != '\*') {

std::cout << "ERROR\_2 " << "comment must start with '(\*' (row:" << currentRow << ", col:" << currentColumn << ")" << std::endl;

errorHappened = true;

return nextLetter;

} else {

token += (char)nextLetter;

}

while (!file.eof()) {

nextLetter = readCharacterFromFile(file);

if (nextLetter == '\*') {

isStartFound = true;

} else if (nextLetter == ')' && isStartFound) {

token += (char)nextLetter;

addToken(token);

token = "";

resetTokenStatus();

return readCharacterFromFile(file);

} else {

isStartFound = false;

}

if (nextLetter == '\n') { currentRow++; currentColumn = 0; }

token += (char)nextLetter;

}

}

bool Lexer::isComment(int letter) {

return letter == '(';

}

void Lexer::scanFile(const char\* filePath) {

std::ifstream file;

file.open(filePath, std::ios::in);

int savedLetter = -1;

while (!file.eof()) {

int letter = savedLetter == -1 ? readCharacterFromFile(file) : savedLetter;

if (isLetter(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseLetter(letter, file);

if (errorHappened) return;

}

if (isOneSeparated(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseOneSeparated(letter, file);

if (errorHappened) return;

}

if (isNumber(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseNumber(letter, file);

if (errorHappened) return;

}

if (isComment(letter)) {

savedLetter = -1;

savedColumn = currentColumn;

savedRow = currentRow;

letter = caseComment(letter, file);

if (errorHappened) return;

}

if (isWhiteSpace(letter)) {

savedColumn = currentColumn;

savedRow = currentRow;

savedLetter = -1;

if (!token.empty()) {

addToken(token);

token = "";

}

resetTokenStatus();

if (letter == '\n') {

currentColumn = 0;

currentRow++;

}

continue;

}

// to prevent endless loop when met undefined letter

if (letter == savedLetter) {

std::cout << "ERROR\_1 " << (char)letter << " code " << letter << " is undefined (row:" << currentRow << ", col:" << currentColumn << ")" << std::endl;

return;

}

savedLetter = letter;

}

}

void Lexer::printScanResult() {

for (auto& result : lexerResultValues) {

result.print();

}

}

**MultiSeparatedToken.h**

#pragma once

class MultiSeparatedToken

{

public:

MultiSeparatedToken(int \_code, int \_pairCode) {

code = \_code;

pairCode = \_pairCode;

}

~MultiSeparatedToken() = default;

inline int getCode() { return code; }

inline int getPairCode() { return pairCode; }

private:

int code;

int pairCode;

};

**LexerResult.h**

#pragma once

#include <string>

#include <iostream>

class LexerResult

{

public:

LexerResult(std::string token, int code, int rowNumber, int columnNumber) {

this->token = token;

this->code = code;

this->columnNumber = columnNumber;

this->rowNumber = rowNumber;

}

~LexerResult() = default;

inline std::string getToken() const { return token; }

inline int getCode() const { return code; }

inline int getRowNumber() const { return rowNumber; }

inline int getColumnNumber() const { return columnNumber; }

void print() {

std::cout << token << ", code=" << code << ", column=" << columnNumber << ", row=" << rowNumber << std::endl;

}

private:

std::string token;

int code;

int rowNumber;

int columnNumber;

};

**TokenStatus.h**

#pragma once

typedef enum TokenStatus {

IDENTIFIER,

RESERVED\_WORD,

ONE\_SEPARATED\_TOKEN,

MULTI\_SEPARATED\_TOKEN,

CONSTANT

};

**Тести**

PROGRAM PR;

VAR A:FLOAT;

VAR C 0 .. 10;

VAR B:FLOAT;

VAR B:FLOAT;

(\*LOLOASDASKLDASDJALKSDJALKDLAKJdLASKdj\*\*\*\*U\*ASKDJAKSDHAD\*\*ASDLJK\*)

(\*BEGIN asdasd

adlaskd;laskdla

alskdalskdjak

\*\*\*\*\*\*\*\*\*\*)

BEGIN

A:=10;

LOOP

C:=0;

A:=C;

A:=C[A];

A:=C[1];

ENDLOOP;

END.

**Вивід програми:**

RESULT

PROGRAM, code=401, column=0, row=0

PR, code=1001, column=8, row=0

;, code=1, column=10, row=0

VAR, code=404, column=2, row=1

A, code=1002, column=6, row=1

:, code=0, column=7, row=1

FLOAT, code=405, column=8, row=1

;, code=1, column=13, row=1

VAR, code=404, column=2, row=2

C, code=1003, column=6, row=2

0, code=501, column=8, row=2

.., code=302, column=10, row=2

10, code=502, column=13, row=2

;, code=1, column=15, row=2

VAR, code=404, column=2, row=3

B, code=1004, column=6, row=3

:, code=0, column=7, row=3

FLOAT, code=405, column=8, row=3

;, code=1, column=13, row=3

VAR, code=404, column=2, row=4

B, code=1004, column=6, row=4

:, code=0, column=7, row=4

FLOAT, code=405, column=8, row=4

;, code=1, column=13, row=4

BEGIN, code=402, column=1, row=10

A, code=1002, column=2, row=11

:=, code=301, column=3, row=11

10, code=502, column=5, row=11

;, code=1, column=7, row=11

LOOP, code=406, column=2, row=12

C, code=1003, column=3, row=13

:=, code=301, column=4, row=13

0, code=501, column=6, row=13

;, code=1, column=7, row=13

A, code=1002, column=3, row=14

:=, code=301, column=4, row=14

C, code=1003, column=6, row=14

;, code=1, column=7, row=14

A, code=1002, column=3, row=15

:=, code=301, column=4, row=15

C, code=1003, column=6, row=15

[, code=2, column=7, row=15

A, code=1002, column=8, row=15

], code=3, column=9, row=15

;, code=1, column=10, row=15

A, code=1002, column=3, row=16

:=, code=301, column=4, row=16

C, code=1003, column=6, row=16

[, code=2, column=7, row=16

1, code=503, column=8, row=16

], code=3, column=9, row=16

;, code=1, column=10, row=16

ENDLOOP, code=407, column=2, row=17

;, code=1, column=9, row=17

END, code=403, column=1, row=18

., code=4, column=4, row=18

**Помилкові тести**

PROGRAM PR;

VAR A$:FLOAT;

VAR A:FLOAT;

VAR C 0 .. 10;

VAR B:FLOAT;

VAR B:FLOAT;

(\*LOLOASDASKLDASDJALKSDJALKDLAKJdLASKdj\*\*\*\*U\*ASKDJAKSDHAD\*\*ASDLJK\*)

(\*BEGIN asdasd

adlaskd;laskdla

alskdalskdjak

\*\*\*\*\*\*\*\*\*\*)

BEGIN

A:=10;

LOOP

C:=0;

A:=C;

A:=C[A];

A:=C[1];

ENDLOOP;

END.

**Вивід програми:**

ERROR\_1 $ code 36 is undefined (row:1, col:7)

RESULT

PROGRAM, code=401, column=0, row=0

PR, code=1001, column=8, row=0

;, code=1, column=10, row=0

VAR, code=404, column=2, row=1

A, code=1002, column=6, row=1

**Тест:**

PROGRAM PR;

VAR A:FLOAT;

VAR C 0 .. 10;

VAR B:FLOAT;

VAR B:FLOAT;

(\*LOLOASDASKLDASDJALKSDJALKDLAKJdLASKdj\*\*\*\*U\*ASKDJAKSDHAD\*\*ASDLJK\*)

(\*BEGIN asdasd

adlaskd;laskdla

alskdalskdjak

\*\*\*\*\*\*\*\*\*\*)

(\*(asdasdasd\*\*\*\*)

BEGIN

A:=1s0;

LOOP

C:=0;

A:=C;

A:=C[A];

A:=C[1];

ENDLOOP;

END.

**Вивід програми:**

ERROR\_3 s code 115 is not number, whitespace or separator (row:12, col:6)

RESULT

PROGRAM, code=401, column=0, row=0

PR, code=1001, column=8, row=0

;, code=1, column=10, row=0

VAR, code=404, column=2, row=1

A, code=1002, column=6, row=1

:, code=0, column=7, row=1

FLOAT, code=405, column=8, row=1

;, code=1, column=13, row=1

VAR, code=404, column=2, row=2

C, code=1003, column=6, row=2

0, code=501, column=8, row=2

.., code=302, column=10, row=2

10, code=502, column=13, row=2

;, code=1, column=15, row=2

VAR, code=404, column=2, row=3

B, code=1004, column=6, row=3

:, code=0, column=7, row=3

FLOAT, code=405, column=8, row=3

;, code=1, column=13, row=3

VAR, code=404, column=2, row=4

B, code=1004, column=6, row=4

:, code=0, column=7, row=4

FLOAT, code=405, column=8, row=4

;, code=1, column=13, row=4

BEGIN, code=402, column=1, row=11

A, code=1002, column=2, row=12

:=, code=301, column=3, row=12

**Тест:**

PROGRAM PR;

VAR A:FLOAT;

VAR C 0 .. 10;

VAR B:FLOAT;

VAR B:FLOAT;

(\*LOLOASDASKLDASDJALKSDJALKDLAKJdLASKdj\*\*\*\*U\*ASKDJAKSDHAD\*\*ASDLJK\*)

(\*BEGIN asdasd

adlaskd;laskdla

alskdalskdjak

\*\*\*\*\*\*\*\*\*\*)

(asdasdasd\*\*\*\*)

BEGIN

LOOP

C:=0;

A:=C;

A:=C[A];

A:=C[1];

ENDLOOP;

END.

**Вивід програми:**

ERROR\_2 comment must start with '(\*' (row:10, col:3)

RESULT

PROGRAM, code=401, column=0, row=0

PR, code=1001, column=8, row=0

;, code=1, column=10, row=0

VAR, code=404, column=2, row=1

A, code=1002, column=6, row=1

:, code=0, column=7, row=1

FLOAT, code=405, column=8, row=1

;, code=1, column=13, row=1

VAR, code=404, column=2, row=2

C, code=1003, column=6, row=2

0, code=501, column=8, row=2

.., code=302, column=10, row=2

10, code=502, column=13, row=2

;, code=1, column=15, row=2

VAR, code=404, column=2, row=3

B, code=1004, column=6, row=3

:, code=0, column=7, row=3

FLOAT, code=405, column=8, row=3

;, code=1, column=13, row=3

VAR, code=404, column=2, row=4

B, code=1004, column=6, row=4

:, code=0, column=7, row=4

FLOAT, code=405, column=8, row=4

;, code=1, column=13, row=4