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

Кафедра системного програмування та спеціалізованих комп’ютерних систем

Лабораторна робота № 1

з дисципліни
«Основи проектування трансляторів»

Тема: «РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА»

Виконав: студент IV курсу
групи КВ-84 ФПМ
Іванюк В.І.
Перевірив:

Київ

2021

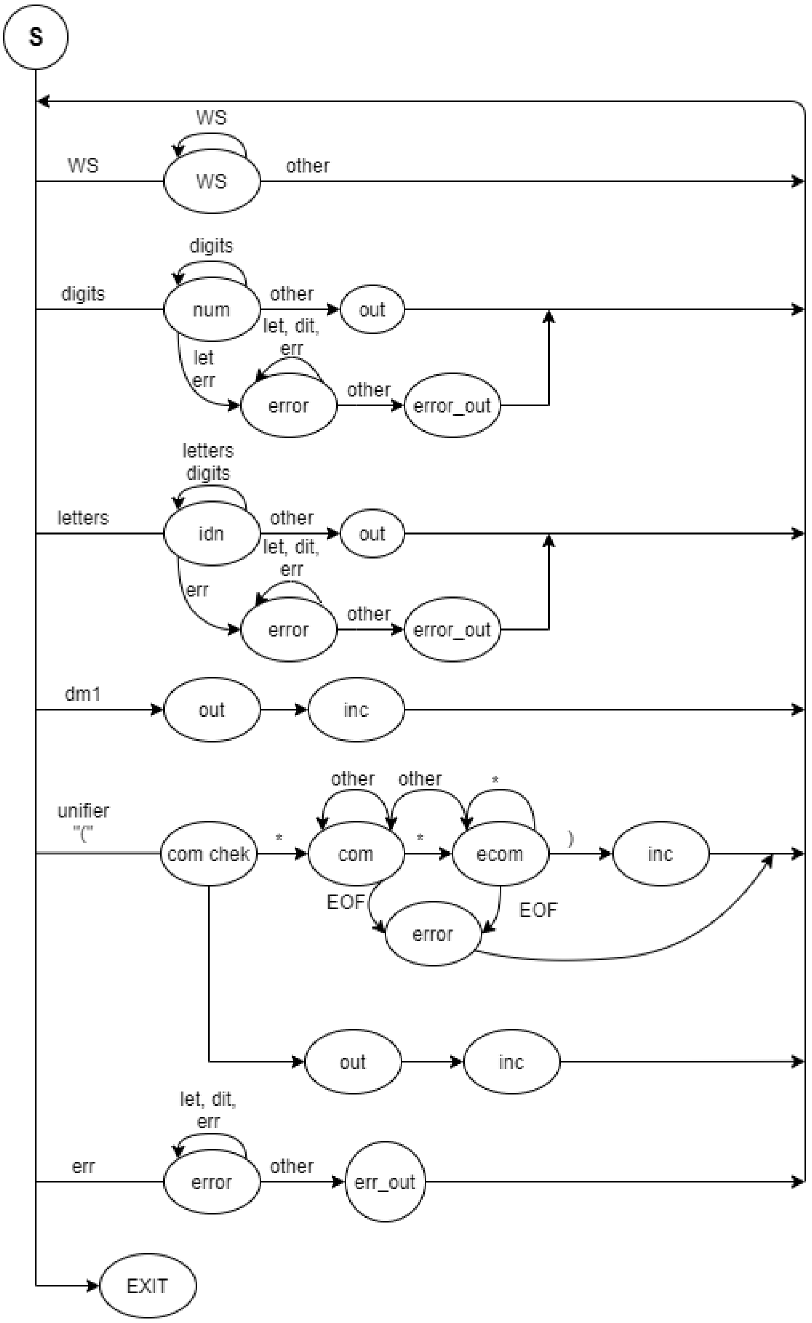
Мета лабораторної роботи

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

Варіант 12

1. < signal - program > -- > < program>
2. < program > -- > PROCEDURE <procedure - identifier> <parameters - list>; <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. < parameters - list > -- > (<variable - identifier> <identifiers - list>) | < empty>
8. < identifiers - list > -- > , <variable - identifier> <identifiers - list> | < empty>
9. < statements - list > -- > <statement> <statements-list> | < empty>
10. < statement > -- > <unsigned - integer> : <statement> | GOTO <unsigned - integer>; | RETURN; | ; | ($ <assembly - insert - file - identifier> $)
11. < variable - identifier > -- > < identifier>
12. < procedure - identifier > -- > < identifier>
13. < assembly - insert - file - identifier > -- > < identifier>
14. < identifier > -- > <letter> < string>
15. < string > -- > <letter><string> | <digit><string> | < empty>
16. < unsigned - integer > -- > <digit> < digits - string>
17. < digits - string > -- > <digit><digits - string> | <empty>
18. < digit > -- > 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
19. < letter > -- > A | B | C | D | ... | Z

Граф автомату, що задає алгоритм ЛА



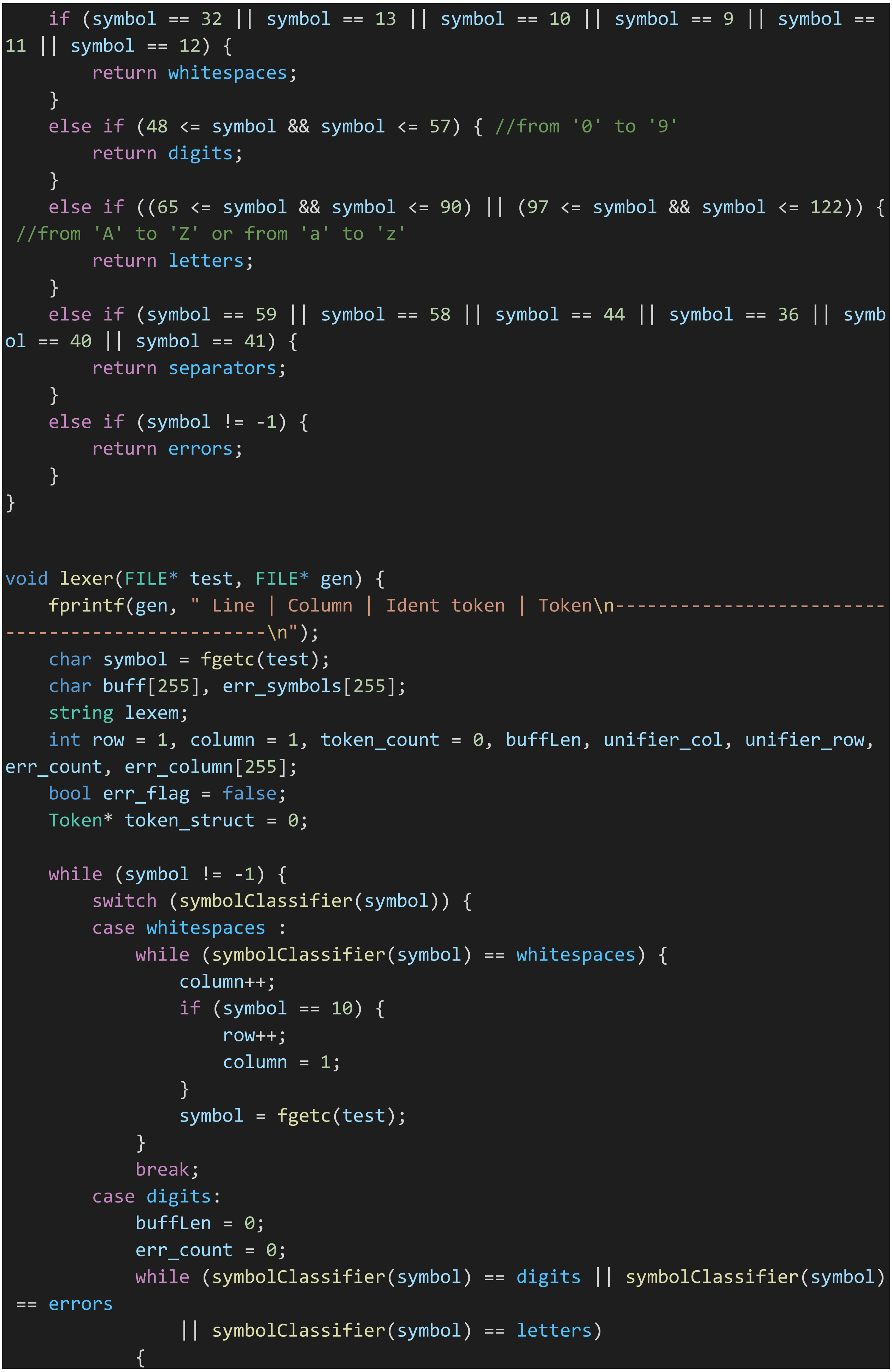
Лістинг програми

OPT_lab1.cpp

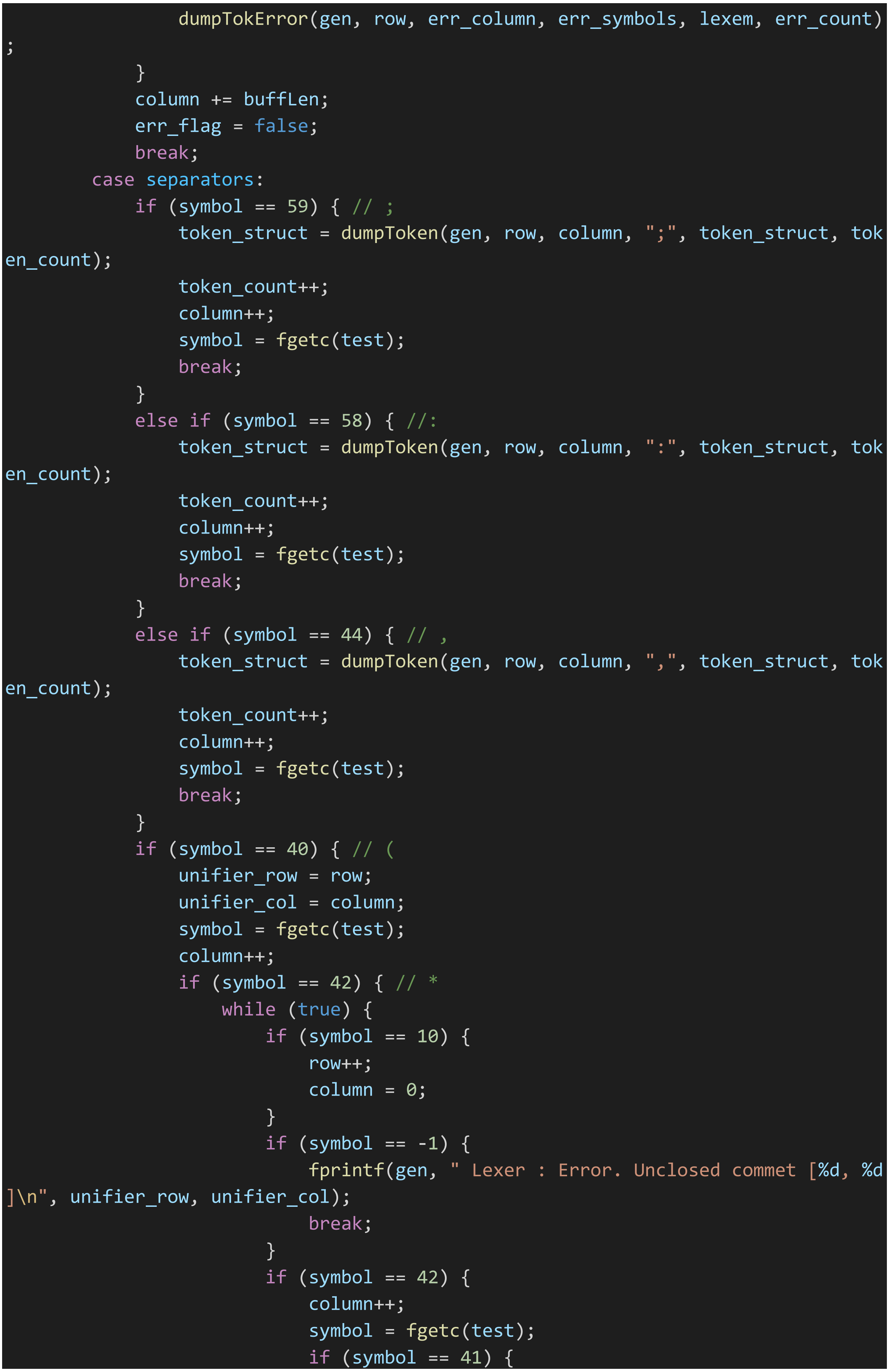
```
#include "LexerGeneration.h"

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Lexer: Invalid number of parameters.");
        return 1;
    }
    else {
        for (int i = 1; i < argc; i++) {
            printf("%s \n", argv[i]);
        }

    }

    FILE* test, * gen;
    char input[30];
    char output[30];
    char inputfile[] = "/input.sig";
    char outputfile[] = "/generated.txt";

    // For Visual Studio 2019
    /*strcpy_s(input, _countof(input), argv[1]);
    strcat_s(input, _countof(input), inputfile);
    strcpy_s(output, _countof(output), argv[1]);
    strcat_s(output, _countof(output), outputfile);
    errno_t err_test, err_gen;
    if ((err_test = fopen_s(&test, input, "r") != 0) || (err_gen = fopen_s(&gen, output, "w") != 0)) {
        return 1;
    }*/

    // For gcc
    strcpy(input, argv[1]);
    strcat(input, inputfile);
    strcpy(output, argv[1]);
    strcat(output, outputfile);
    if (((test = fopen(input, "r")) == NULL) || ((gen = fopen(output, "w")) == NULL)) {
        return 1;
    }


    else {
        lexer(test, gen);

        fclose(test);
        fclose(gen);
    }

    return 0;

}
```

LexerGeneration.cpp

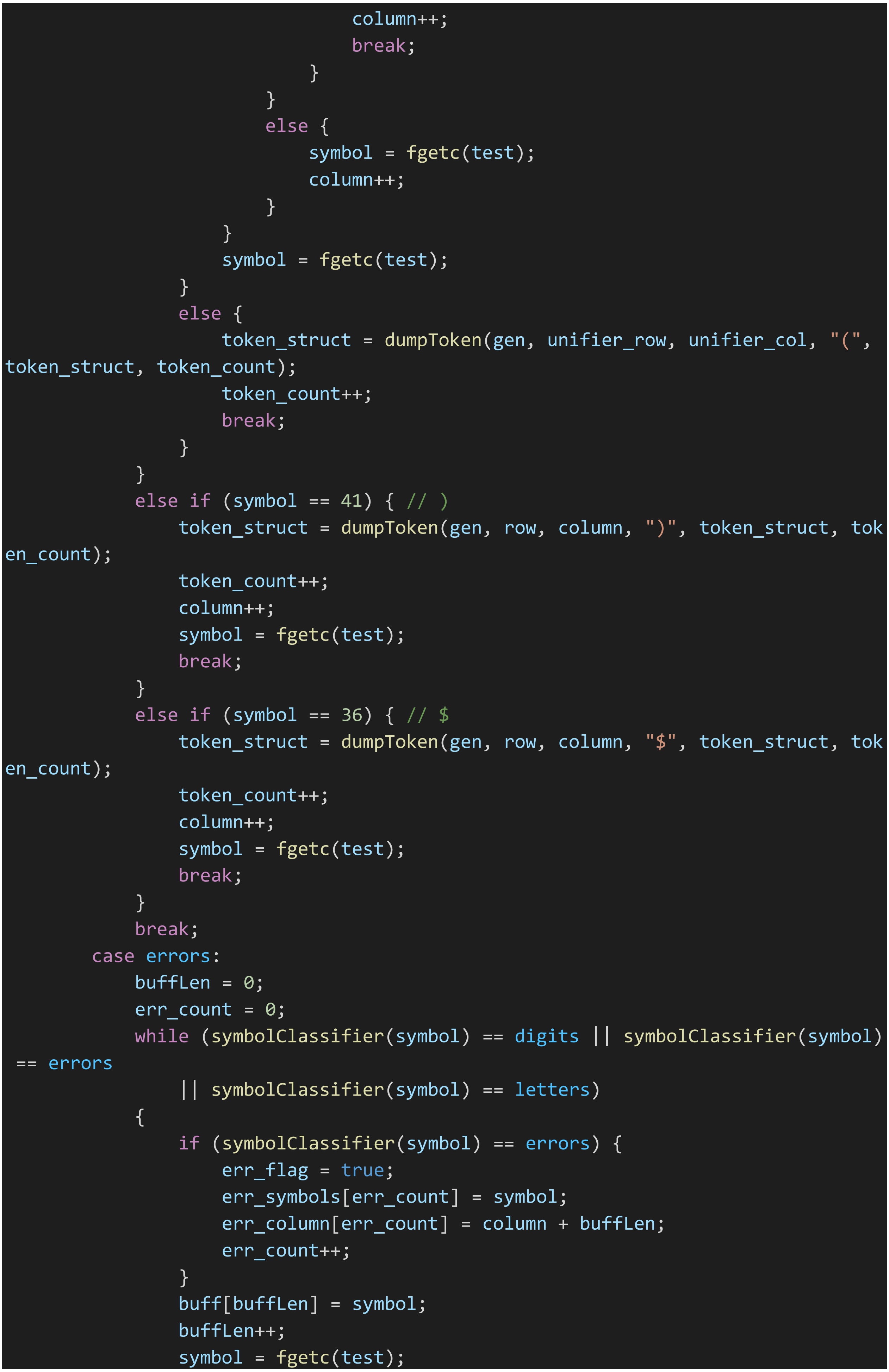
```
#include "LexerGeneration.h"


Token* dumpToken(FILE* generated, int row, int column, string token, Token* tokenStruct, const int count) {
    tokenStruct = AddToken(row, column, findID(token), token, tokenStruct, count);
    fprintf(generated, " %4d | %6d | %11d | %s\n", row, column, findID(token), token.c_str());
    return tokenStruct;
}

void dumpTokError(FILE* generated, int row, int *column, char *err_symb, string token, int count) {
    fprintf(generated, " Lexer : Error. Illegam symbol : ");
    for (int i = 0; i < count; i++) {
        fprintf(generated, "'%c'[%d, %d] ", err_symb[i], row, column[i]);
    }
    fprintf(generated, "in %s\n", token.c_str());
}

void dumpLexError(FILE* generated, int row, int column, string token) {
    fprintf(generated, " Lexer : Error. Illegam symbol : '%s'[%d, %d]\n", token.c_str(), row, column);
}

Token* AddToken(int row, int column, int id, string token, Token* tokenStruct, const  int count) {
    if (count == 0) {
        tokenStruct = new Token[count + 1];
    }
    else {
        Token* tmpToken = new Token[count + 1];
        for (int i = 0; i < count; i++) {
            tmpToken[i] = tokenStruct[i];
        }
        delete[] tokenStruct;

        tokenStruct = tmpToken;
    }
    tokenStruct[count].row = row;
    tokenStruct[count].column = column;
    tokenStruct[count].id = id;
    tokenStruct[count].value = token;

    return tokenStruct;
}

void showTokens(const Token* tokenStruct, const int count) {
    for (int i = 0; i < count; i++) {
        cout << tokenStruct[i].row << " | " << tokenStruct[i].column << " | " << tokenStruct[i].id << " | " << tokenStruct[i].value << endl;
    }
}

int symbolClassifier(char symbol) {
```



```
    if (symbol == 32 || symbol == 13 || symbol == 10 || symbol == 9 || symbol == 11 || symbol == 12) {
        return whitespaces;
    }
    else if (48 <= symbol && symbol <= 57) { //from '0' to '9'
        return digits;
    }
    else if ((65 <= symbol && symbol <= 90) || (97 <= symbol && symbol <= 122)) { //from 'A' to 'Z' or from 'a' to 'z'
        return letters;
    }
    else if (symbol == 59 || symbol == 58 || symbol == 44 || symbol == 36 || symbol == 40 || symbol == 41) {
        return separators;
    }
    else if (symbol != -1) {
        return errors;
    }
}


void lexer(FILE* test, FILE* gen) {
    fprintf(gen, " Line | Column | Ident token | Token\n-------------------------------------------------\n");
    char symbol = fgetc(test);
    char buff[255], err_symbols[255];
    string lexem;
    int row = 1, column = 1, token_count = 0, buffLen, unifier_col, unifier_row, err_count, err_column[255];
    bool err_flag = false;
    Token* token_struct = 0;

    while (symbol != -1) {
        switch (symbolClassifier(symbol)) {
        case whitespaces :
            while (symbolClassifier(symbol) == whitespaces) {
                column++;
                if (symbol == 10) {
                    row++;
                    column = 1;
                }
                symbol = fgetc(test);
            }
            break;
        case digits:
            buffLen = 0;
            err_count = 0;
            while (symbolClassifier(symbol) == digits || symbolClassifier(symbol) == errors
                || symbolClassifier(symbol) == letters)
            {
                if (symbolClassifier(symbol) == errors || symbolClassifier(symbol) == letters) {
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                }
                buff[buffLen] = symbol;
                buffLen++;
                symbol = fgetc(test);
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (err_flag == false) {
                token_struct = dumpToken(gen, row, column, lexem, token_struct, token_count);
                token_count++;
            }
            else {
                dumpTokError(gen, row, err_column, err_symbols, lexem, err_count);
            }
            column += buffLen;
            err_flag = false;
            break;
        case letters:
            buffLen = 0;
            err_count = 0;
            while (symbolClassifier(symbol) == digits || symbolClassifier(symbol) == errors
                || symbolClassifier(symbol) == letters)
            {
                if (symbolClassifier(symbol) == errors) {
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                }
                buff[buffLen] = symbol;
                buffLen++;
                symbol = fgetc(test);
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (err_flag == false) {
                token_struct = dumpToken(gen, row, column, lexem, token_struct, token_count);
                token_count++;
            }
            else {
```



```
                dumpTokError(gen, row, err_column, err_symbols, lexem, err_count)
;
            }
            column += buffLen;
            err_flag = false;
            break;
        case separators:
            if (symbol == 59) { // ;
                token_struct = dumpToken(gen, row, column, ";", token_struct, tok
en_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            else if (symbol == 58) { //:
                token_struct = dumpToken(gen, row, column, ":", token_struct, tok
en_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            else if (symbol == 44) { // ,
                token_struct = dumpToken(gen, row, column, ",", token_struct, tok
en_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            if (symbol == 40) { // (
                unifier_row = row;
                unifier_col = column;
                symbol = fgetc(test);
                column++;
                if (symbol == 42) { // *
                    while (true) {
                        if (symbol == 10) {
                            row++;
                            column = 0;
                        }
                        if (symbol == -1) {
                            fprintf(gen, " Lexer : Error. Unclosed commet [%d, %d
]\n", unifier_row, unifier_col);
                            break;
                        }
                        if (symbol == 42) {
                            column++;
                            symbol = fgetc(test);
                            if (symbol == 41) {
```



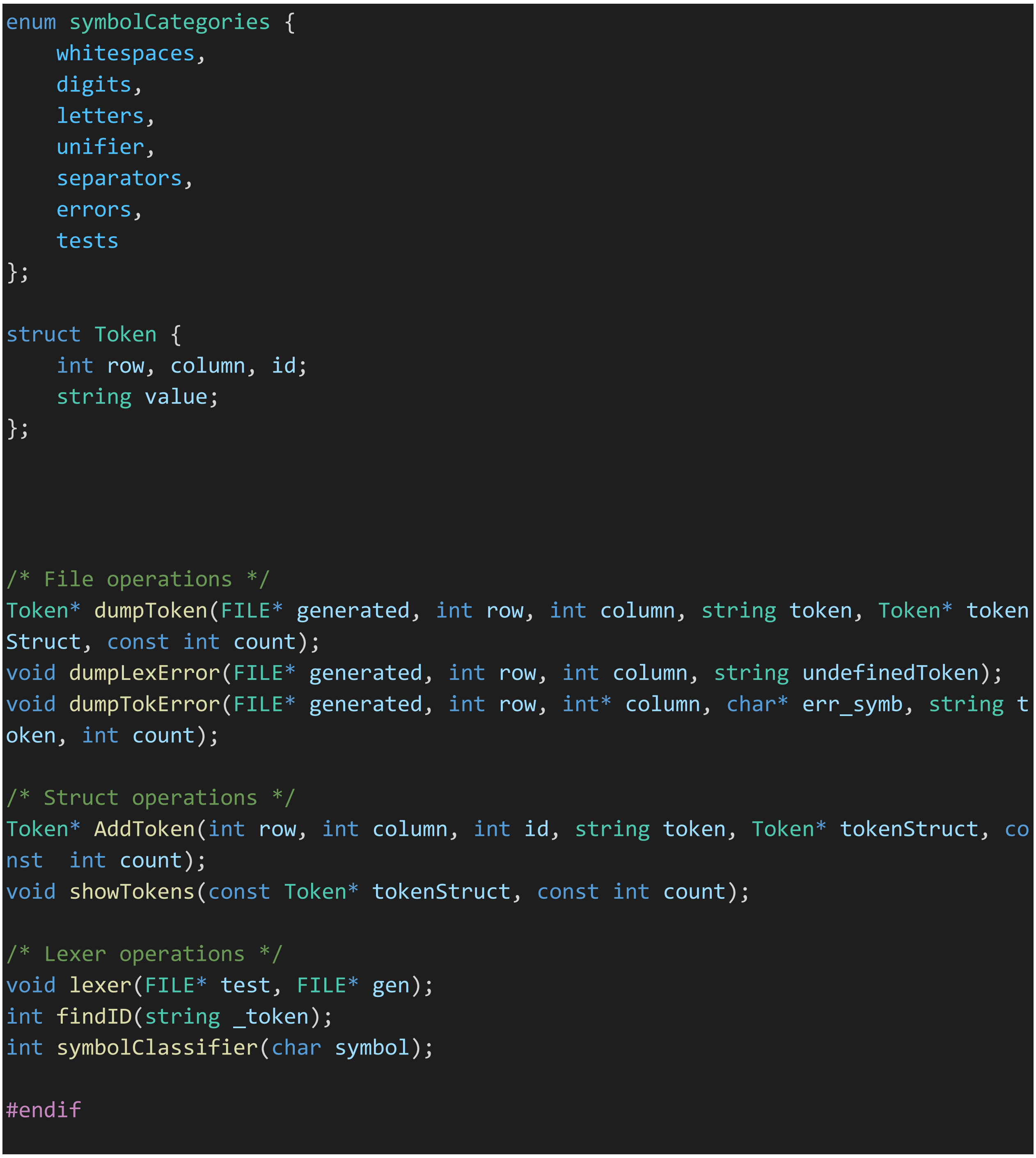
```
                                column++;
                                break;
                            }
                        }
                        else {
                            symbol = fgetc(test);
                            column++;
                        }
                    }
                    symbol = fgetc(test);
                }
                else {
                    token_struct = dumpToken(gen, unifier_row, unifier_col, "(", 
token_struct, token_count);
                    token_count++;
                    break;
                }
            }
            else if (symbol == 41) { // )
                token_struct = dumpToken(gen, row, column, ")", token_struct, tok
en_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            else if (symbol == 36) { // $
                token_struct = dumpToken(gen, row, column, "$", token_struct, tok
en_count);
                token_count++;
                column++;
                symbol = fgetc(test);
                break;
            }
            break;
        case errors:
            buffLen = 0;
            err_count = 0;
            while (symbolClassifier(symbol) == digits || symbolClassifier(symbol)
 == errors
                || symbolClassifier(symbol) == letters)
            {
                if (symbolClassifier(symbol) == errors) {
                    err_flag = true;
                    err_symbols[err_count] = symbol;
                    err_column[err_count] = column + buffLen;
                    err_count++;
                }
                buff[buffLen] = symbol;
                buffLen++;
                symbol = fgetc(test);
            }
            buff[buffLen] = '\0';
            lexem = string(buff);
            if (buffLen > 1)
                dumpTokError(gen, row, err_column, err_symbols, lexem, err_count)
;
            else
                dumpLexError(gen, row, column, lexem);
            column += buffLen;
            err_flag = false;
            break;
        }
    }
    showTokens(token_struct, token_count);
}
```

LexerTables.cpp

```
#include "LexerGeneration.h"

int ident_count = 1001;
int const_count = 501;

map <string, int> kwrd = {
  {"PROCEDURE", 401},
  {"BEGIN", 402},
  {"END", 403},
  {"LABEL", 404},
  {"GOTO", 405},
  {"RETURN", 406}
};

map <string, int> sep = {
    {";", 59},
    {",", 44},
    {":", 58},
    {"(", 40},
    {")", 41},
    {"$", 36}
};

map <string, int> ident;
map <string, int> _const;
int findID(string _token) {
    Token token;
    token.value = _token;
    map<string, int>::iterator iter;

    if (symbolClassifier(token.value[0]) == letters) {
        if (kwrd.count(token.value) == 1) {
            iter = kwrd.find(token.value);
            token.id = iter->second;
        }
        else if (ident.count(token.value) == 0) {
            ident.insert(pair<string, int>(token.value, ident_count));
            token.id = ident_count;
            ident_count++;
        }
        else {
            iter = ident.find(token.value);
            token.id = iter->second;
        }
    }
    else if (symbolClassifier(token.value[0]) == digits) {
        if (_const.count(token.value) == 0) {
            _const.insert(pair<string, int>(token.value, const_count));
            token.id = const_count;
            const_count++;
        }
        else {
            iter = _const.find(token.value);
            token.id = iter->second;
        }
    }
    else if (sep.count(token.value) == 1) {
        iter = sep.find(token.value);
        token.id = iter->second;
    }

    return token.id;
}
```

LexerGeneration.h

```
#pragma once
#ifndef LEXERGENERATION_H
#define LEXERGENERATION_H

#include <iostream>
#include <string>

#include <cctype>
#include <algorithm>
#include <vector>
#include <typeinfo>
#include <cstring>
#include <stdio.h>
#include <map>

using namespace std;
```



```
enum symbolCategories {
    whitespaces,
    digits,
    letters,
    unifier,
    separators,
    errors,
    tests
};

struct Token {
    int row, column, id;
    string value;
};




/* File operations */
Token* dumpToken(FILE* generated, int row, int column, string token, Token* token
Struct, const int count);
void dumpLexError(FILE* generated, int row, int column, string undefinedToken);
void dumpTokError(FILE* generated, int row, int* column, char* err_symb, string t
oken, int count);

/* Struct operations */
Token* AddToken(int row, int column, int id, string token, Token* tokenStruct, co
nst  int count);
void showTokens(const Token* tokenStruct, const int count);

/* Lexer operations */
void lexer(FILE* test, FILE* gen);
int findID(string _token);
int symbolClassifier(char symbol);

#endif
```

Контрольні приклади

Test01

Input.sig

```
PROCEDURE proc;
proc BEGIN:
LABEL label1:
($ asmFile $)
GOTO label1;
l#g%7b
&hjk
(*!*****)
(**) (* *)
( *)
```

```
(*
comment*)
END
(*
```

Generated.txt

```
 Line | Column | Ident token | Token
-------------------------------------------------
    1 |      1 |         401 | PROCEDURE
    1 |     11 |        1001 | proc
    1 |     15 |          59 | ;
    2 |      1 |        1001 | proc
    2 |      6 |         402 | BEGIN
    2 |     11 |          58 | :
    3 |      1 |         404 | LABEL
    3 |      7 |        1002 | label1
    3 |     13 |          58 | :
    4 |      1 |          40 | (
    4 |      2 |          36 | $
    4 |      4 |        1003 | asmFile
    4 |     12 |          36 | $
    4 |     13 |          41 | )
    5 |      1 |         405 | GOTO
    5 |      6 |        1002 | label1
    5 |     12 |          59 | ;
 Lexer : Error. Illegam symbol : '#'[6, 2] '%'[6, 4] in l#g%7b
 Lexer : Error. Illegam symbol : '&'[7, 1] in &hjk
   10 |      1 |          40 | (
 Lexer : Error. Illegam symbol : '*'[10, 3]
   10 |      4 |          41 | )
   13 |      1 |         403 | END
 Lexer : Error. Unclosed commet [14, 1]
```

Test02

Input.sig

```
PROCEDURE proc;
proc BEGIN:
LABEL label1:
( var1 var2
() (var1 var1);
($ asmFile $) ($ $)
( ( heu $)
($ asm )
END
($ asm
```

Generated.txt

```
 Line | Column | Ident token | Token
-------------------------------------------------
    1 |      1 |         401 | PROCEDURE
    1 |     11 |        1001 | proc
    1 |     15 |          59 | ;
    2 |      1 |        1001 | proc
    2 |      6 |         402 | BEGIN
    2 |     11 |          58 | :
    3 |      1 |         404 | LABEL
    3 |      7 |         1002 | label1
    3 |     13 |           58 | :
    4 |      1 |           40 | (
    4 |      3 |         1003 | var1
    4 |      8 |         1004 | var2
    5 |      1 |           40 | (
    5 |      2 |           41 | )
    5 |      4 |           40 | (
    5 |      5 |         1003 | var1
    5 |     10 |         1003 | var1
    5 |     14 |           41 | )
    5 |     15 |           59 | ;
    6 |      1 |           40 | (
    6 |      2 |           36 | $
    6 |      4 |         1005 | asmFile
    6 |     12 |           36 | $
    6 |     13 |           41 | )
    6 |     15 |           40 | (
    6 |     16 |           36 | $
    6 |     18 |           36 | $
    6 |     19 |           41 | )
    7 |      1 |           40 | (
    7 |      3 |           40 | (
    7 |      5 |         1006 | heu
    7 |      9 |           36 | $
    7 |     10 |           41 | )
    8 |      1 |           40 | (
    8 |      2 |           36 | $
    8 |      4 |         1007 | asm
    8 |      8 |           41 | )
    9 |      1 |          403 | END
   10 |      1 |           40 | (
   10 |      2 |           36 | $
   10 |      4 |         1007 | asm
```

Test03

Input.sig

```
PROCEDURE proc;
proc BEGIN:
LABEL label1:
(var1 var2 var3)
($ asmFile $)
105 1c%
END
```

Generated.txt

```
 Line | Column | Ident token | Token
-------------------------------------------------
    1 |      1 |          401 | PROCEDURE
    1 |     11 |         1001 | proc
    1 |     15 |           59 | ;
    2 |      1 |         1001 | proc
    2 |      6 |          402 | BEGIN
    2 |     11 |           58 | :
    3 |      1 |          404 | LABEL
    3 |      7 |         1002 | label1
    3 |      13 |          58 | :
    4 |       1 |          40 | (
    4 |       2 |        1003 | var1
    4 |       7 |        1004 | var2
    4 |      12 |        1005 | var3
    4 |      16 |          41 | )
    5 |       1 |          40 | (
    5 |       2 |          36 | $
    5 |       4 |        1006 | asmFile
    5 |      12 |          36 | $
    5 |      13 |          41 | )
    6 |       1 |         501 | 105
 Lexer : Error. Illegam symbol : 'c'[6, 6] '%'[6, 7] in 1c%
    7 |       1 |         403 | END
```

Test04

Input.sig

```
PROCEDURE proc;
proc BEGIN:
LABEL la%bel1:
($ asm_File $)
% &var1;
( &var )
END
```

Generated.txt

```
 Line | Column | Ident token | Token
-------------------------------------------------
    1 |       1 |         401 | PROCEDURE
    1 |      11 |        1001 | proc
    1 |      15 |          59 | ;
    2 |       1 |        1001 | proc
    2 |       6 |         402 | BEGIN
    2 |      11 |          58 | :
    3 |       1 |         404 | LABEL
 Lexer : Error. Illegam symbol : '%'[3, 9] in la%bel1
    3 |      14 |          58 | :
    4 |       1 |          40 | (
    4 |       2 |          36 | $
 Lexer : Error. Illegam symbol : '_'[4, 7] in asm_File
    4 |      13 |          36 | $
    4 |      14 |          41 | )
 Lexer : Error. Illegam symbol : '%'[5, 1]
 Lexer : Error. Illegam symbol : '&'[5, 3] in &var1
    5 |       8 |          59 | ;
    6 |       1 |          40 | (
 Lexer : Error. Illegam symbol : '&'[6, 3] in &var
    6 |       8 |          41 | )
    7 |       1 |         403 | END
```