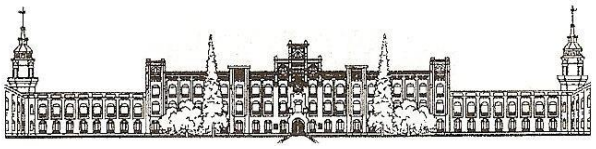


Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Лабораторна робота №1

з дисципліни «ОСНОВИ ПРОЕКТУВАННЯ ТРАНСЛЯТОРІВ»

«РОЗРОБКА ЛЕКСИЧНОГО АНАЛІЗАТОРА»

Виконав студент групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: _____

Київ 2024

Постановка задачі

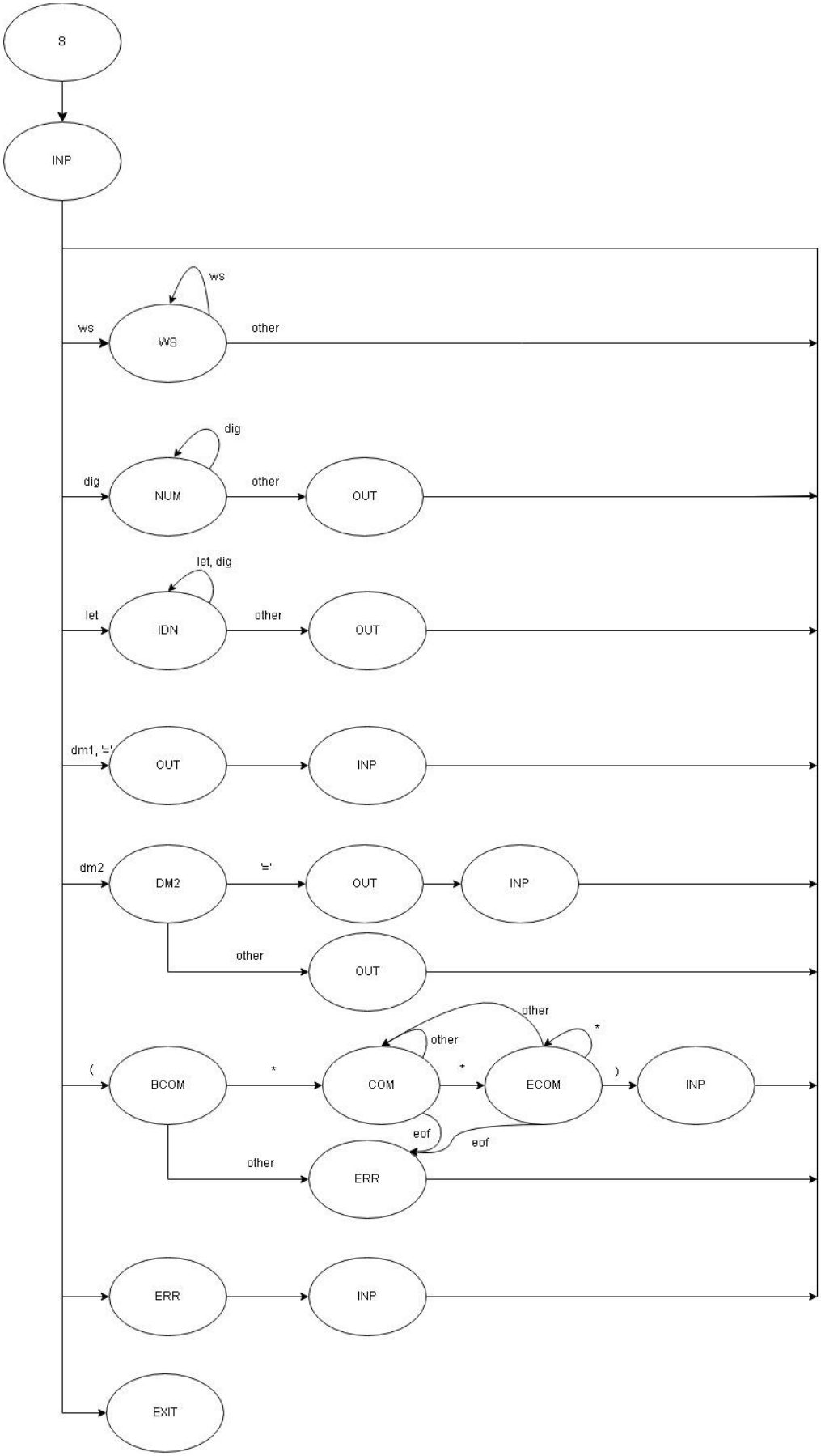
Розробити програму лексичного аналізатора (ЛА) для підмножини мови програмування SIGNAL. Лексичний аналізатор має забезпечувати наступні дії:

- видалення (пропускання) пробільних символів: пробіл (код ASCII 32), повернення каретки (код ASCII 13); перехід на новий рядок (код ASCII 10), горизонтальна та вертикальна табуляція (коди ASCII 9 та 11), перехід на нову сторінку (код ASCII 12);
- згортання ключових слів;
- згортання багато-символьних роздільників (якщо передбачаються граматикою варіанту);
- згортання констант із занесенням до таблиці значення та типу константи (якщо передбачаються граматикою варіанту);
- згортання ідентифікаторів;
- видалення коментарів, заданих у вигляді (<текст коментаря>);
- формування рядка лексем з інформацією про позиції лексем;
- заповнення таблиць ідентифікаторів та констант інформацією, отриманою під час згортки лексем;
- виведення повідомлень про помилки.

Граматика за варіантом 21

<signal-program> --> <program>
<program> --> PROGRAM <procedure-identifier> ;
<block>.
<block> --> <declarations> BEGIN <statements-list> END
<declarations> --> <constant-declarations>
<constant-declarations> --> CONST <constantdeclarations-list> |
<empty>
<constant-declarations-list> --> <constantdeclaration> <constant-
declarations-list> |
<empty>
<constant-declaration> --> <constant-identifier> =
<constant>;
<statements-list> --> <statement> <statements-list> |
<empty>
<statement> --> CASE <expression> OF <alternativeslist> ENDCASE ;
<alternatives-list> --> <alternative> <alternativeslist> |
<empty>
<alternative> --> <expression> : /<statements-list>\
<expression> --> <summand> <summands-list> |
- <summand> <summands-list>
<summands-list> --> <add-instruction> <summand>
<summands-list> |
<empty>
<add-instruction> --> + |
-
<summand> --> <variable-identifier> |
<unsigned-integer>
<constant> --> <unsigned-integer>
<variable-identifier> --> <identifier>
<constant-identifier> --> <identifier>
<procedure-identifier> --> <identifier>
<identifier> --> <letter><string>
<string> --> <letter><string> |
<digit><string> |
<empty>
<unsigned-integer> --> <digit><digits-string>
<digits-string> --> <digit><digits-string> |
<empty>
<digit> --> **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
<letter> --> A | B | C | D | ... | Z

Граф автомату



token.h

```
#ifndef TOKEN_H
#define TOKEN_H
struct token {
  unsigned int row;
  unsigned int col;
  unsigned long int code;
  char *_data;
  unsigned int dataSize;
};
typedef struct token Token;

extern Token *_tokens;
extern unsigned long int tokenCount;

Token create_token(unsigned int row, unsigned int col, unsigned long int code,
                   char _data[], unsigned int dataSize);
void add_to_tokens(Token token);
#endif
```

token.c

```
#include "token.h"
#include "error.h"
#include <stddef.h>
#include <stdlib.h>

Token *_tokens = NULL;
unsigned long int tokenCount = 0;

void add_to_tokens(Token token) {
  tokenCount++;
  _tokens = (Token *)realloc(_tokens, (tokenCount) * sizeof(Token));
  if (_tokens == NULL) {
    add_to_errors(create_error_with_linecolumn(errorCount + 1, 0,
                                               "Cannot reallocate *_tokens",
                                               true, token.row, token.col));
  } else {
    _tokens[tokenCount - 1] = token;
  }
}

Token create_token(unsigned int row, unsigned int col, unsigned long int code,
                   char _data[], unsigned int dataSize) {
  char *__data = malloc((dataSize + 1) * sizeof(char));
  for (unsigned int i = 0; i < dataSize; i++)
    __data[i] = _data[i];
  __data[dataSize] = '\0';
  Token token = {row - 1, col - 1, code, __data, dataSize};
  return token;
}
```

out.h

```
#ifndef OUT_H
#define OUT_H
#include "cli.h"
#include "error.h"
#include "lexer.h"
#include "token.h"
void print_params();
void print_error(Error error);
```

```
void print_errors();
void print_lexer();
void print_token(Token token);
void print_tokens();
void out_file_lexer();
void print_file_out();
void out_file_errors();
#endif
```

out.c

```
#include "out.h"
#include <stdio.h>

void print_params() {
  printf("Input file: %s\n", params._input_file);
  printf("Output file: %s\n", params._output_file);
  if (params.verbose)
    printf("Verbose mode enabled\n");
}
void print_error(Error error) {
  char *critical = "Warning";
  short int state = error.state;
  if (error.critical)
    critical = "Error";
  if (state == 0)
    if (error.hasLineColumn)
      printf("#%ld|%s(Lexer)| Line->%d, Column->%d |: %s\n", error.number,
             critical, error.row, error.col, error._error_message);
    else
      printf("#%ld|%s(Lexer): %s\n", error.number, critical,
             error._error_message);
  else if (state)
    printf("#%ld|%s(File IO): %s\n", error.number, critical,
           error._error_message);
  else
    printf("#%ld|%s(Unknown): %s\n", error.number, critical,
           error._error_message);
}

void get_error(Error error, FILE *__output_file) {
  char *critical = "Warning";
  short int state = error.state;
  if (error.critical)
    critical = "Error";
  if (state == 0)
    if (error.hasLineColumn)
      fprintf(__output_file, "#%ld|%s(Lexer)| Line->%d, Column->%d |: %s\n",
              error.number, critical, error.row, error.col,
              error._error_message);
    else
      fprintf(__output_file, "#%ld|%s(Lexer): %s\n", error.number, critical,
              error._error_message);
  else if (state)
    fprintf(__output_file, "#%ld|%s(File IO): %s\n", error.number, critical,
            error._error_message);
  else
    fprintf(__output_file, "#%ld|%s(Unknown): %s\n", error.number, critical,
            error._error_message);
}

void print_errors() {
  for (unsigned long int i = 0; i < errorCount; i++) {
    print_error(_errors[i]);
  }
```

```
}
void print_lexer() {
  printf("Current state: %u\n", lexer.state);
  printf("Current buffer: %s\n", lexer._buffer);
  printf("Current row: %u\n", lexer.row);
  printf("Current col: %u\n", lexer.col);
  printf("Current symbol: %c\n", lexer.symbol);
  printf("Current symbol type: %d\n", lexer.symbolType);
}
void print_token(Token token) {
  printf("[%u][%u] %lu: %s\n", token.row, token.col, token.code, token._data);
}
void print_tokens() {
  for (unsigned long int i = 0; i < tokenCount; i++) {
    print_token(_tokens[i]);
  }
}

void out_file_lexer() {
  FILE *__output_file;
  __output_file = fopen(params._output_file, "w");
  if (__output_file == NULL) {
    add_to_errors(create_error_without_linecolumn(
        errorCount + 1, -1, "Cannot write to output file", true));
  } else {
    fprintf(__output_file,
            "|Line  |Column|Code  |Data  \n+------+------+------+------\n");
    for (unsigned long int i = 0; i < tokenCount; i++) {
      fprintf(__output_file, "|%6d|%6d|%6ld|%s\n", _tokens[i].row,
              _tokens[i].col, _tokens[i].code, _tokens[i]._data);
    }
  }
  if (params.verbose) {
    out_file_errors(__output_file);
  }
  fclose(__output_file);
}

void print_file_out() {
  FILE *__output_file;
  __output_file = fopen(params._output_file, "r");
  if (__output_file == NULL) {
    add_to_errors(create_error_without_linecolumn(
        errorCount + 1, -1, "Cannot open output file for reading", true));
  } else {
    for (char c = (char)getc(__output_file); c != EOF;
         c = (char)getc(__output_file))
      printf("%c", c);
  }
}

void out_file_errors(FILE *__output_file) {
  fprintf(__output_file, "ERRORS:\n");
  for (unsigned int i = 0; i < errorCount; i++) {
    get_error(_errors[i], __output_file);
  }
}
```

main.c

```
#include "out.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
  proc_cli(argc, argv);
  if (gotError) {
```

```
    print_errors();
    return -1;
  } else {
    proc_lexer(params._input_file);
  }
  if (params.verbose) {
    print_errors();
    out_file_lexer();
  } else {
    out_file_lexer();
    print_file_out();
  }
  return 0;
}
```

lexer.h

```
#ifndef LEXER_H
#define LEXER_H
#include "cli.h"
#include <stddef.h>
struct lexer {
  unsigned int state;
  /*
  0: whitespace - Reading next tokek(whitespace)
  1: number - Reading next token(number)
  11: add - Reading + and -
  2: identifier - Reading next token(identifier)
  3: delimiter1 - Reading next token(delimeter1)
  4: delimiter2 - Reading next token(delimeter2)
  51: comment begin - Reading ('('), BCOM
  52: comment confirm - Reading token('*'), COMCON
  53: comment end - Reading token(')'), ECOM
  6: ERR - error
  7: EXIT - exit state
  8: START - start state
  */

  // Buff work
  char *_buffer;
  unsigned int bufferSize;
  unsigned short int row;
  unsigned short int col;
  char symbol;
  unsigned short int symbolType;
  bool com;
};
typedef struct lexer Lexer;

extern Lexer lexer;

void proc_lexer(char *_input_file);

#endif
```

lexer.c

```
#include "lexer.h"
#include "error.h"
#include "token.h"
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
Lexer lexer = {0, NULL, 0, 1, 1, '\0', 6, false};

unsigned int lastConst = 0;
unsigned long int lastIdentifier = 0;
char **identifiers;

void add_buffer_symbol() {
  if (lexer.state != 52) {
    lexer._buffer =
        (char *)realloc(lexer._buffer, lexer.bufferSize * sizeof(char));
    if (lexer._buffer == NULL) {
      add_to_errors(create_error_with_linecolumn(errorCount + 1, 0,
                                                 "Cannot resize *buff", true,
                                                 lexer.row, lexer.col));
    }
    lexer._buffer[lexer.bufferSize] = lexer.symbol;
    lexer.bufferSize++;
  }
}

void clean_buffer() {
  lexer._buffer = NULL;
  lexer.bufferSize = 0;
}

/*
0 - ws: whitespace(and etc.)| ASCII 8->13, 32
1 - dig: numbers| ASCII 48->57
11 - add: + or -
2 - let: identifiers and keywords| ASCII 65->90, 97->122
3 - dm1: delimeters first type
4 - dm2: delimeters second type(for 2 symbols in token)
51 - com_beg: comment begin '('
52 - com_confirm: comment confirm '*'
53 - com_end: comment end ')'
6 - err: error symbols| ASCII 0->7, 127 or any not listed here
7 - eof: end-of-file symbol| not ASCII symbol
*/
unsigned short int symbol_type(char symbol) {
  unsigned short int category = 6;
  if ((symbol > 7 && symbol < 14) || symbol == 32)
    category = 0;
  else if (symbol > 47 && symbol < 58)
    category = 1;
  else if (symbol > 64 && symbol < 91)
    category = 2;
  else if (symbol == '.' || symbol == ';' || symbol == '[' || symbol == ']' ||
           symbol == '=' || symbol == '+' || symbol == '-')
    category = 3;
  else if (symbol == '=')
    category = 31;
  else if (symbol == ':' || symbol == '<' || symbol == '>')
    category = 4;
  else if (symbol == '(')
    category = 5;
  else if (symbol == EOF)
    category = 7;
  else
    category = 6;

  return category;
}

unsigned short int is_keyword() {
  char *_verify[10] = {"PROGRAM", "VAR", "BEGIN",   "END",     "CONST",
```

```
                      "CASE",    "OF",  "ENDCASE", "INTEGER", "FLOAT"};
  for (unsigned short int i = 0; i < 10; i++) {
    if (!strcmp(lexer._buffer, _verify[i]))
      return i + 1;
  }
  return 0;
}

unsigned short int get_code_dm1() {
  char _verify[10] = {'+', '-', ':', '<', '>', '=', '.', ';', '[', ']'};
  for (unsigned short i = 0; i < 10; i++) {
    if (lexer._buffer[0] == _verify[i])
      return (unsigned short int)lexer._buffer[0];
  }
  return 0;
}
unsigned short int get_code_dm2() {
  char _verify[3] = {'<', '>', ':'};
  if (lexer._buffer[1] == '=') {
    for (unsigned short i = 0; i < 3; i++) {
      if (lexer._buffer[0] == _verify[i])
        return i + 1;
    }
  }
  return get_code_dm1();
}

unsigned long int get_code() {
  unsigned long int base = 0;
  switch (lexer.state) {
  case 1:
    base = 500;
    base += lastConst + 1;
    break;
  case 2:
    if (is_keyword()) {
      base = 400;
      base += is_keyword();
    } else {
      base = 1000;
      if (identifiers != NULL) {
        for (unsigned long int i = 0; i < lastIdentifier; i++) {
          if (!strcmp(lexer._buffer, identifiers[i])) {
            return base + i + 1;
          }
        }
      }
      identifiers =
          (char **)realloc(identifiers, (lastIdentifier + 1) * sizeof(char *));
      if (identifiers == NULL) {
        add_to_errors(create_error_with_linecolumn(
            errorCount + 1, 0, "Cannot resize **identifiers", true, lexer.row,
            lexer.col));
      }
      identifiers[lastIdentifier] = lexer._buffer;
      lastIdentifier++;
      return base + lastIdentifier;
    }
    break;
  case 3:
  case 11:
    base = 0 + get_code_dm1();
    break;
  case 4:
    base = get_code_dm2();
```

```
    if (lexer._buffer[1] == '=')
      base += 300;
    break;

  default:
    add_to_errors(create_error_without_linecolumn(
        errorCount + 1, 0, "Impossible for get_code()", true));
    return 0;
  };
  return base;
}

void inp(FILE *__input_file) {
  lexer.symbol = (char)fgetc(__input_file);
  if (lexer.symbol == '\n') {
    lexer.row++;
    lexer.col = 1;
  } else {
    if (lexer.symbol == '\t')
      lexer.col += 4;
    else {
      if (lexer.symbol == EOF) {
        lexer.state = 7;
      }
      lexer.col++;
    }
  }
  lexer.symbolType = symbol_type(lexer.symbol);
}

void got_ws(FILE *__input_file) {
  do {
    inp(__input_file);
  } while (lexer.symbolType == 0);
  lexer.state = 0;
}
void got_dig(FILE *__input_file) {
  unsigned int row = lexer.row;
  unsigned int col = lexer.col;
  lexer.state = 1;
  do {
    add_buffer_symbol();
    inp(__input_file);
  } while (lexer.symbolType == 1);

  add_to_tokens(
      create_token(row, col, get_code(), lexer._buffer, lexer.bufferSize));
  clean_buffer();
}
void got_let(FILE *__input_file) {
  unsigned int row = lexer.row;
  unsigned int col = lexer.col;
  lexer.state = 2;
  do {
    add_buffer_symbol();
    inp(__input_file);
  } while (lexer.symbolType == 1 || lexer.symbolType == 2);

  add_to_tokens(
      create_token(row, col, get_code(), lexer._buffer, lexer.bufferSize));
  clean_buffer();
}
void got_dm1(FILE *__input_file) {
  unsigned int row = lexer.row;
  unsigned int col = lexer.col;
```

```
  lexer.state = 3;
  add_buffer_symbol();
  inp(__input_file);
  add_to_tokens(
      create_token(row, col, get_code(), lexer._buffer, lexer.bufferSize));
  clean_buffer();
}

void got_dm2(FILE *__input_file) {
  unsigned int row = lexer.row;
  unsigned int col = lexer.col;
  lexer.state = 4;
  add_buffer_symbol();
  inp(__input_file);
  if (lexer.symbolType == 3) {
    add_buffer_symbol();
    inp(__input_file);
  }
  add_to_tokens(
      create_token(row, col, get_code(), lexer._buffer, lexer.bufferSize));
  clean_buffer();
}

void got_com(FILE *__input_file, unsigned int row, unsigned int col);
void got_ecom(FILE *__input_file, unsigned int row, unsigned int col);

void got_com_beg(FILE *__input_file) {
  lexer.state = 51;
  unsigned int row = lexer.row;
  unsigned int col = lexer.col;
  inp(__input_file);
  if (lexer.symbol == '*') {
    lexer.com = true;
    got_com(__input_file, row, col);
  } else {
    add_to_errors(create_error_with_linecolumn((errorCount + 1), 0,
                                               "No * after (", true, row, col));
    inp(__input_file);
  }
}

void got_com(FILE *__input_file, unsigned int row, unsigned int col) {
  inp(__input_file);
  if (lexer.symbol == '*') {
    got_ecom(__input_file, row, col);
  } else {
    if (lexer.symbolType == 7) {
      add_to_errors(create_error_with_linecolumn(
          errorCount + 1, 0, "Not closed comment", true, row, col));
      inp(__input_file);
    } else {
      got_com(__input_file, row, col);
    }
  }
}

void got_ecom(FILE *__input_file, unsigned int row, unsigned int col) {
  inp(__input_file);
  if (lexer.symbol == ')') {
    inp(__input_file);
    lexer.state = 0;
    lexer.com = false;
  } else {
    if (lexer.symbol == '*')
      got_ecom(__input_file, row, col);
```

```
    else {
      if (lexer.symbolType == 7) {
        add_to_errors(create_error_with_linecolumn(
            errorCount + 1, 0, "Not closed comment", true, row, col));
        inp(__input_file);
      } else
        got_com(__input_file, row, col);
    }
  }
}

void proc_lexer(char *_input_file) {
  FILE *__input_file;
  __input_file = fopen(_input_file, "r");
  if (__input_file == NULL) {
    add_to_errors(create_error_without_linecolumn(
        (errorCount + 1), -1, "Cannot open input file.", true));
  } else {
    lexer.state = 8;
    inp(__input_file);
    do {
      switch (lexer.symbolType) {
      case 0:
        got_ws(__input_file);
        break;
      case 1:
        got_dig(__input_file);
        break;
      case 2:
        got_let(__input_file);
        break;
      case 3:
      case 31:
        got_dm1(__input_file);
        break;
      case 4:
        got_dm2(__input_file);
        break;
      case 5:
        got_com_beg(__input_file);
        break;
      case 6:
        if (lexer.symbol == '*' || lexer.symbol == ')')
          add_to_errors(create_error_with_linecolumn(
              errorCount + 1, 0, "Comment is not openned or already closed",
              false, lexer.row, lexer.col));
        else
          add_to_errors(create_error_with_linecolumn((errorCount + 1), 0,
                                                     "Got error symbol", true,
                                                     lexer.row, lexer.col));
        inp(__input_file);
        break;
      case 7:
        lexer.state = 7;
        break;
      default:
        add_to_errors(create_error_without_linecolumn(
            errorCount + 1, 0, "Impossible if rrly, unknown category", true));
        break;
      };
    } while (lexer.state != 7);
  }
}
```

error.h

```
#ifndef ERROR_H
#define ERROR_H
#include <stdbool.h>

struct error {
  // -1 -> Means that error does not exist
  long int number;
  // 0 -> Lexer, -1 -> File access, -2 -> Memory access
  short int state;
  char *_error_message;
  // true -> Error, false -> Warning
  bool critical;
  bool hasLineColumn;
  unsigned int row;
  unsigned int col;
};
typedef struct error Error;

extern Error *_errors;
extern unsigned int errorCount;
extern bool gotError;
extern bool gotWarning;

Error create_error_without_linecolumn(long int number, short int state,
                                      char *_error_message, bool critical);
Error create_error_with_linecolumn(long int number, short int state,
                                   char *_error_message, bool critical,
                                   unsigned int row, unsigned int col);
Error create_error_def();
void add_to_errors(Error error);
bool has_critical();

#endif
```

error.c

```
#include "error.h"
#include <stdio.h>
#include <stdlib.h>

Error *_errors = NULL;
unsigned int errorCount = 0;
bool gotError = false;
bool gotWarning = false;

bool has_critical() {
  for (unsigned int i = 0; i < errorCount; i++) {
    if (_errors[i].critical)
      return true;
  }
  return false;
}

Error create_error_without_linecolumn(long int number, short int state,
                                      char *_error_message, bool critical) {
  Error error = {number, state, _error_message, critical, false, 0, 0};
  return error;
}

Error create_error_with_linecolumn(long int number, short int state,
                                   char *_error_message, bool critical,
                                   unsigned int row, unsigned int col) {
```

```
  Error error = {number, state, _error_message, critical, true, row, col};
  return error;
}

Error create_error_def() {
  Error error = {-1, -2, "", false, false, 0, 0};
  return error;
}

void add_to_errors(Error error) {
  errorCount++;
  _errors = (Error *)realloc(_errors, (errorCount) * sizeof(Error));
  if (_errors == NULL) {
    exit(EXIT_FAILURE);
  } else {
    _errors[errorCount - 1] = error;
    if (error.critical)
      gotError = true;
    else
      gotWarning = true;
  }
}
```

cli.h

```
#ifndef CLI_H
#define CLI_H
#include "error.h"
#include <stdbool.h>
struct params {
  char *_input_file;
  char *_output_file;
  bool verbose;
};
typedef struct params Params;

extern Params params;

void proc_cli(int argc, char *argv[]);

#endif
```

cli.c

```
#include "cli.h"
#include "error.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

Params params = {NULL, NULL, false};

void check_file_access(char *_file, bool inputFile) {
  if (access(_file, F_OK) != 0) {
    if (inputFile) {
      add_to_errors(create_error_without_linecolumn(
          errorCount + 1, -1, "Missing access to input file", true));
    } else {
      add_to_errors(create_error_without_linecolumn(
          errorCount + 1, -1, "File for output does not exist, creating...",
          false));
    }
  }
```

```
}

void check_file_missing(char *_file) {
  FILE *_fp;
  _fp = fopen(_file, "w");

  if (_fp == NULL) {
    add_to_errors(create_error_without_linecolumn(
        errorCount + 1, -1, "Cannot create/open output file", true));
  }
  fclose(_fp);
}

void proc_cli(int argc, char *argv[]) {
  if (argc == 2) {
    params._input_file = argv[1];
  } else {
    for (int i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-f") == 0 && i + 1 < argc) {
        params._input_file = argv[i + 1];
        i++;
      } else if (strcmp(argv[i], "-o") == 0 && i + 1 < argc) {
        params._output_file = argv[i + 1];
        i++;
      } else if (strcmp(argv[i], "-v") == 0) {
        params.verbose = 1;
      }
    }
  }

  if (params._input_file == NULL) {
    add_to_errors(create_error_without_linecolumn(
        errorCount + 1, -1, "Input filename is empty.", true));
  } else {
    check_file_access(params._input_file, true);
    check_file_access(params._output_file, false);
    check_file_missing(params._output_file);
  }
}
```

True_test.sig

```
(**)
PROGRAM TEST02;
    VAR
        ITEM1 : INTEGER;
        ITEM2 : FLOAT;
    BEGIN
        ITEM2 := 20;
        ITEM3 := ITEM2;
        ITEM1 := ITEM3 + ITEM2;
        ITEM2 := 20;
    END. (*(*End of file**)
```

```
t3ry4@iamhost ~/repos/mine/t3ry444y-git/OPT-lab1/src $ cat output
|Line  |Column|Code  |Data
+------+------+------+------
|     1|     1|   401|PROGRAM
|     1|     9|  1001|TEST02
|     1|    15|    59|;
|     2|     5|   402|VAR
|     3|     9|  1002|ITEM1
|     3|    15|    58|:
|     3|    17|   409|INTEGER
|     3|    24|    59|;
|     4|     9|  1003|ITEM2
|     4|    15|    58|:
|     4|    17|   410|FLOAT
|     4|    22|    59|;
|     5|     5|   403|BEGIN
|     6|     9|  1003|ITEM2
|     6|    15|   303|:=
|     6|    18|   501|20
|     6|    20|    59|;
|     7|     9|  1004|ITEM3
|     7|    15|   303|:=
|     7|    18|  1003|ITEM2
|     7|    23|    59|;
|     8|     9|  1002|ITEM1
|     8|    15|   303|:=
|     8|    18|  1004|ITEM3
|     8|    24|    43|+
|     8|    26|  1003|ITEM2
|     8|    31|    59|;
|     9|     9|  1003|ITEM2
|     9|    15|   303|:=
|     9|    18|   501|20
|     9|    20|    59|;
|    10|     5|   404|END
|    10|     8|    46|.
ERRORS:
t3ry4@iamhost ~/repos/mine/t3ry444y-git/OPT-lab1/src $
```

False_test.sig

```
(**)
PROGRAM TEST02;
    VAR
        ITEM1 : INTEGER;
        ITEM2 : FLOAT;
    BEGIN
        # some error
        ErrOr
        ITEM2 := 20;
        ITEM3 := ITEM2;
        ITEM1 := ITEM3 + ITEM2;
        ITEM2 := 20;
    END. (*(*End of file**)**)
```

```
|Line  |Column|Code  |Data
+------+------+------+------
|     1|     1|   401|PROGRAM
|     1|     9|  1001|TEST02
|     1|    15|    59|;
|     2|     5|   402|VAR
|     3|     9|  1002|ITEM1
|     3|    15|    58|:
|     3|    17|   409|INTEGER
|     3|    24|    59|;
|     4|     9|  1003|ITEM2
|     4|    15|    58|:
|     4|    17|   410|FLOAT
|     4|    22|    59|;
|     5|     5|   403|BEGIN
|     7|     9|  1004|E
|     7|    12|  1005|O
|     8|     9|  1003|ITEM2
|     8|    15|   303|:=
|     8|    18|   501|20
|     8|    20|    59|;
|     9|     9|  1006|ITEM3
|     9|    15|   303|:=
|     9|    18|  1003|ITEM2
|     9|    23|    59|;
|    10|     9|  1002|ITEM1
|    10|    15|   303|:=
|    10|    18|  1006|ITEM3
|    10|    24|    43|+
|    10|    26|  1003|ITEM2
|    10|    31|    59|;
|    11|     9|  1003|ITEM2
|    11|    15|   303|:=
|    11|    18|   501|20
|    11|    20|    59|;
|    12|     5|   404|END
|    12|     8|    46|.
ERRORS:
#1|Error(Lexer)| Line->7, Column->10 |: Got error symbol
#2|Error(Lexer)| Line->7, Column->12 |: Got error symbol
#3|Error(Lexer)| Line->7, Column->13 |: Got error symbol
#4|Error(Lexer)| Line->7, Column->14 |: Got error symbol
#5|Error(Lexer)| Line->7, Column->15 |: Got error symbol
#6|Error(Lexer)| Line->7, Column->17 |: Got error symbol
#7|Error(Lexer)| Line->7, Column->18 |: Got error symbol
#8|Error(Lexer)| Line->7, Column->19 |: Got error symbol
#9|Error(Lexer)| Line->7, Column->20 |: Got error symbol
#10|Error(Lexer)| Line->7, Column->21 |: Got error symbol
#11|Error(Lexer)| Line->8, Column->11 |: Got error symbol
#12|Error(Lexer)| Line->8, Column->12 |: Got error symbol
#13|Error(Lexer)| Line->8, Column->14 |: Got error symbol
#14|Warning(Lexer)| Line->13, Column->29 |: Comment is not openned or already closed
#15|Warning(Lexer)| Line->13, Column->30 |: Comment is not openned or already closed
#16|Warning(Lexer)| Line->13, Column->31 |: Comment is not openned or already closed
t3ry4@iamhost ~/repos/mine/t3ry444y-git/OPT-lab1/src $
```