

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>

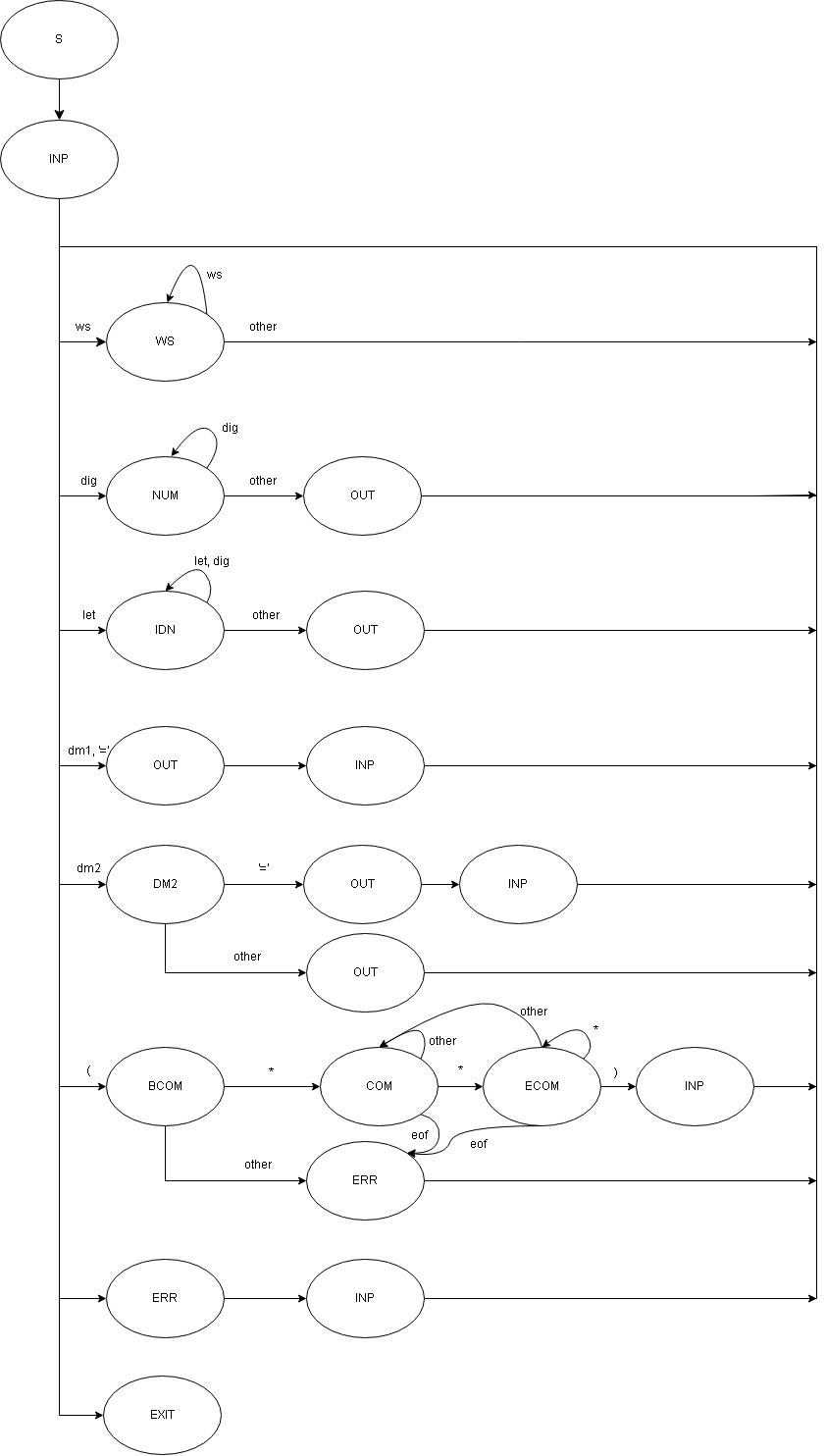
<digits-string> --> <digit><digits-string> |

<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*

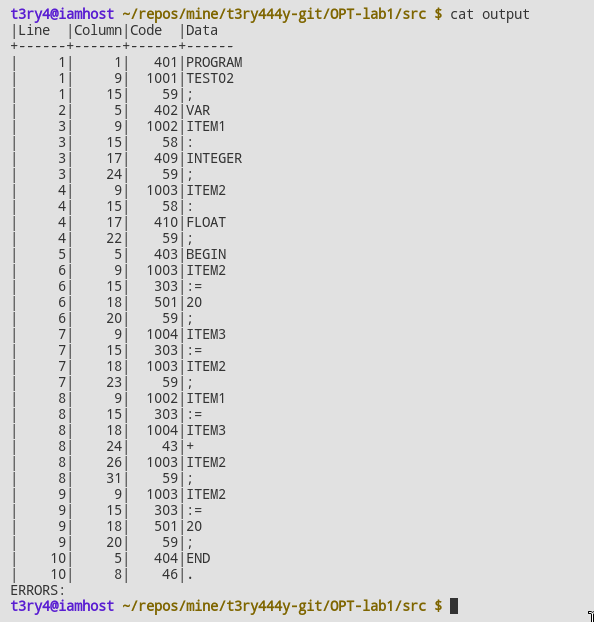
*ITEM2 := 20;*

*ITEM3 := ITEM2;*

*ITEM1 := ITEM3 + ITEM2;*

*ITEM2 := 20;*

*END. (\*(\*End of file\*\*)*

**

*False\_test.sig*

*(\*\*)*

*PROGRAM TEST02;*

*VAR*

*ITEM1 : INTEGER;*

*ITEM2 : FLOAT;*

*BEGIN*

*# some error*

*ErrOr*

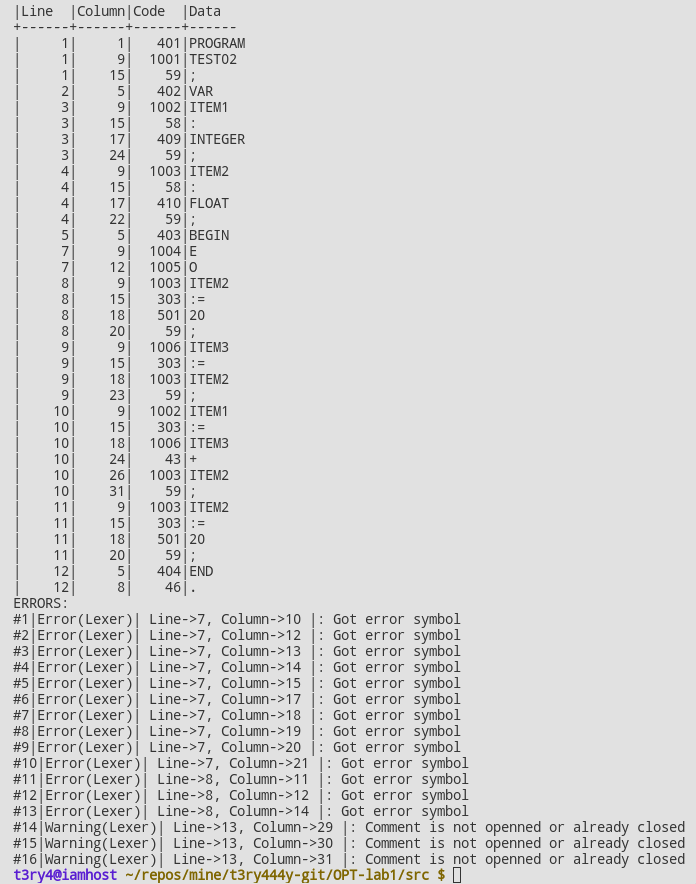
*ITEM2 := 20;*

*ITEM3 := ITEM2;*

*ITEM1 := ITEM3 + ITEM2;*

*ITEM2 := 20;*

*END. (\*(\*End of file\*\*)\*\*)*

**