

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. Ігоря Сікорського»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

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

**Лабораторна робота №1**

З дисципліни

***«Основи проектування трансляторів»***

**Тема: Розробка лексичного аналізатора**

Виконав:

Студент групи КВ-11

Парієнко Віктор

Перевірив: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Київ 2024**

**Варіант 19**

**Постановка задачі**

1. Розробити програму лексичного аналізатора (ЛА) для підмножини мови програмування SIGNAL.
2. 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;

**Граматика за варіантом 19**

**<signal-program>** --> <program>

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

**<block>** --> <variable-declarations> **BEGIN** <statements-list> **END**

**<variable-declarations>** --> **VAR** <declarations-list> | <empty>

**<declarations-list>** --> <declaration> <declarations-list> | <empty>

**<declaration>** --><variable-identifier>**:**<attribute> **;**

**<attribute>** --> **INTEGER** | **FLOAT**

**<statements-list>** --> <statement> <statements-list> | <empty>

**<statement>** --> <condition-statement> **ENDIF ;**

**<condition-statement>** --> <incomplete-condition-statement><alternative-part>

**<incomplete-condition-statement>** --> **IF** <conditional-expression> **THEN <statements-list>**

**<alternative-part>** --> **ELSE** <statements-list> | <empty>

**<conditional-expression>** --> <expression> **=** <expression>

**<expression>** --> <variable-identifier> | <unsigned-integer>

**<variable-identifier>** --> <identifier>

**<procedure-identifier>** --> <identifier>

**<identifier>** --> <letter><string>

**<string>** --> <letter><string> | <digit><string> | <empty>

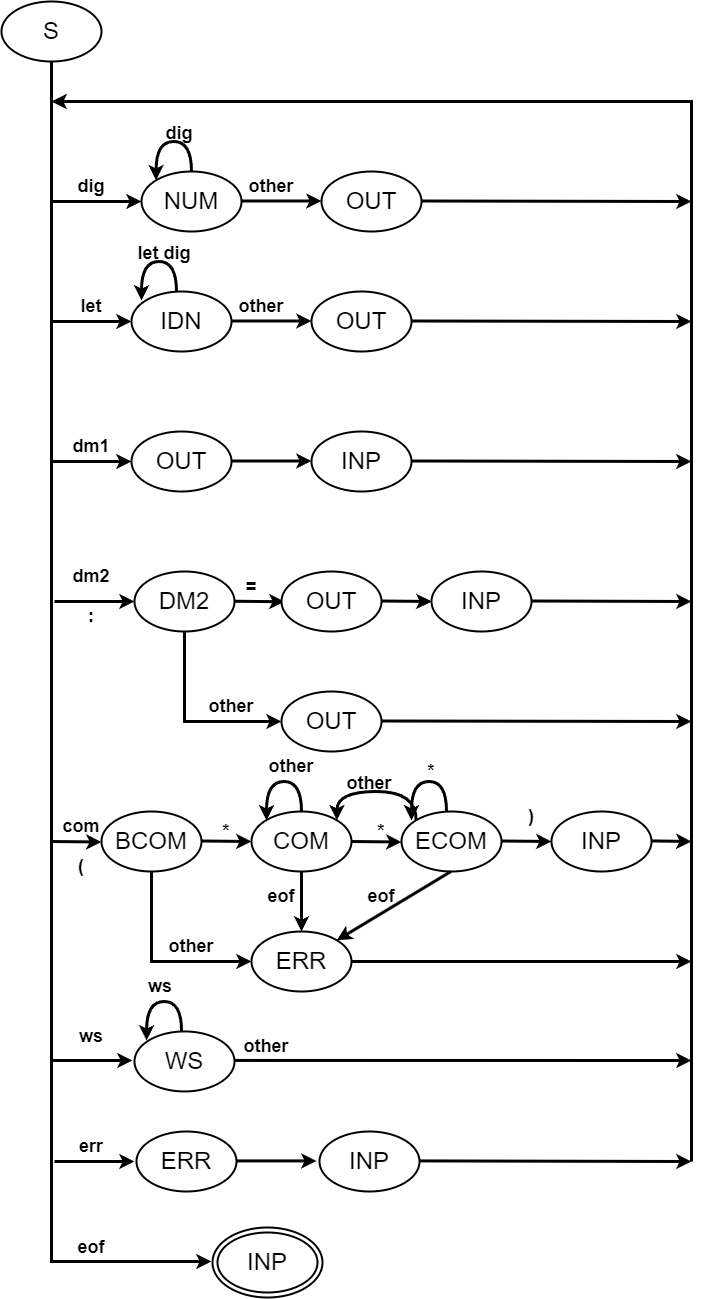
**<unsigned-integer>** --> <digit><digits-string>

**<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**

****

**Лістинг коду програми**

*Log.h*

#ifndef LOG\_H\_

#define LOG\_H\_

#include <iostream>

#define CRIMSON "\033[31m"

#define LIME "\033[32m"

#define LEMON "\033[33m"

#define AZURE "\033[34m"

#define MAGENTA "\033[35m"

#define TEAL "\033[36m"

#define RESET "\033[0m"

#define UNDERLINE "\033[4m"

#define INVERSE "\033[7m"

class Log

{

public:

static void Init();

static Log\* GetLogger() { return s\_Logger; }

static void SetSeparator(char separator) { GetLogger()->m\_Separator = separator; }

template <typename... Types>

void State(Types... objects)

{

SetColor(LEMON);

PrintTime();

Print(objects...);

SetColor(RESET);

}

template <typename... Types>

void Trace(Types... objects)

{

PrintTime();

Print(objects...);

}

template <typename... Types>

void Error(Types... objects)

{

SetColor(CRIMSON);

PrintTime();

Print(objects...);

SetColor(RESET);

}

private:

template <typename T>

void Print(const T& obj)

{

*std*::*cout* << obj << "\n";

}

template <typename T, typename... Types>

void Print(const T& obj, const Types&... objects)

{

*std*::*cout* << obj << m\_Separator;

Print(objects...);

}

void SetColor(const char\* color);

void PrintTime();

private:

static Log\* s\_Logger;

char m\_Separator;

};

#ifdef COMPILE\_DEBUG

#define LOG\_STATE(...) Log::GetLogger()->State(\_\_VA\_ARGS\_\_);

#define LOG\_ERROR(...) Log::GetLogger()->Error(\_\_VA\_ARGS\_\_);

#define LOG\_TRACE(...) Log::GetLogger()->Trace(\_\_VA\_ARGS\_\_);

#else

#define LOG\_STATE

#define LOG\_ERROR

#define LOG\_TRACE

#endif

#endif /\* LOG\_H\_ \*/

*Log.cpp*

#include "Log.h"

#include <ctime>

#include <string>

Log\* Log::s\_Logger;

void Log::Init()

{

s\_Logger = new Log;

s\_Logger->m\_Separator = ' ';

LOG\_STATE("Initialized Log!");

}

void Log::SetColor(const char\* color)

{

*std*::*cout* << color;

}

void Log::PrintTime()

{

*time\_t* now = *time*(0);

*tm*\* lctime = *localtime*(&now);

auto h = lctime->*tm\_hour*;

auto m = lctime->*tm\_min*;

auto s = lctime->*tm\_sec*;

*std*::*cout* << '[' << (h > 9 ? "\0" : "0") << h << ':' << (m > 9 ? "\0" : "0") << m << ':' << (s > 9 ? "\0" : "0") << s << "] ";

}

*CompilerInterface.h*

#ifndef COMPILERINTERFACE\_H\_

#define COMPILERINTERFACE\_H\_

#include "Errors/ErrorHandler.h"

#include "Lexer/Lexer.h"

#include "Log.h"

#include <memory>

class CompilerInterface

{

public:

virtual void OutErrors() = 0;

virtual void OutTokens() = 0;

virtual void OutIdentifiersTable() = 0;

virtual void OutConstantsTable() = 0;

virtual void OutKeywordsTable() = 0;

virtual void OutOptions() = 0;

virtual void UsageHint(char\* name) = 0;

void SetErrorHandler(*std*::*shared\_ptr*<ErrorHandler> errorHandler);

void SetLexerData(*std*::*shared\_ptr*<LexerData> lexerData);

void SetInfoFileName(const *std*::*string*& fileName);

void SetOutToFileEnabled(bool option);

protected:

*std*::*shared\_ptr*<ErrorHandler> m\_ErrorHandler;

*std*::*shared\_ptr*<LexerData> m\_LexerData;

*std*::*string* m\_InfoFileName;

bool bOutToFile;

*std*::*ofstream* m\_Ofs;

};

class CLInterface : public CompilerInterface

{

public:

~CLInterface();

virtual void OutErrors() override;

virtual void OutTokens() override;

virtual void OutIdentifiersTable() override;

virtual void OutConstantsTable() override;

virtual void OutKeywordsTable() override;

virtual void OutOptions() override;

virtual void UsageHint(char\* name) override;

private:

void DisplayTable(const *std*::*unordered\_map*<*std*::*string*, *uint32\_t*>& table, const *std*::*string*& tableHeader);

};

#endif /\* COMPILERINTERFACE\_H\_ \*/

*CompilerInterface.cpp*

#include "CompilerInterface.h"

#include "Lexer/Token.h"

#include "Log.h"

#include <iostream>

#include <iomanip>

#include <vector>

#include <fstream>

*std*::*string* GetInstigatorColor(EErrorInstigator inst)

{

switch (inst)

{

case EErrorInstigator::Lexer:

return TEAL;

default:

return RESET;

}

}

void CompilerInterface::SetErrorHandler(*std*::*shared\_ptr*<ErrorHandler> errorHandler)

{

m\_ErrorHandler = errorHandler;

}

void CompilerInterface::SetLexerData(*std*::*shared\_ptr*<LexerData> lexerData)

{

m\_LexerData = lexerData;

}

void CompilerInterface::SetInfoFileName(const *std*::*string*& fileName)

{

m\_InfoFileName = fileName;

m\_Ofs.*open*(m\_InfoFileName);

if (!m\_Ofs)

{

LOG\_ERROR("Error opening file:", m\_InfoFileName);

}

}

void CompilerInterface::SetOutToFileEnabled(bool option)

{

bOutToFile = option;

}

CLInterface::~CLInterface()

{

m\_Ofs.*close*();

}

void CLInterface::OutErrors()

{

if (m\_ErrorHandler->GetErrors()->*empty*())

return;

*std*::*cout* << CRIMSON <<"============ Error List: ============\n\n" << RESET;

for (Error& error : \*m\_ErrorHandler->GetErrors())

{

*std*::*cout* << "[";

*std*::*cout* << GetInstigatorColor(error.GetEnumInstigator());

*std*::*cout* << error.GetInstigator() << RESET << "] ";

if (error.GetLine() != "0")

*std*::*cout* << TEAL << error.GetLine() << RESET << "," << TEAL << error.GetPosition() << RESET << "): ";

*std*::*cout* << CRIMSON << error.GetType() << RESET << ": " << error.GetMessage() << "\n";

}

*std*::*cout* << CRIMSON << "=====================================\n\n" << RESET;

if (!bOutToFile)

return;

m\_Ofs << "============ Error List: ============\n\n";

for (Error& error : \*m\_ErrorHandler->GetErrors())

{

m\_Ofs << "[" << error.GetInstigator() << "] ";

if (error.GetLine() != "0")

m\_Ofs << "(" << error.GetLine() << "," << error.GetPosition() << "): ";

m\_Ofs << error.GetType()<< ": " << error.GetMessage() << "\n";

}

m\_Ofs << "=====================================\n\n";

}

void CLInterface::OutTokens()

{

const *uint32\_t* lineWidth = 4;

const *uint32\_t* posWidth = 4;

const *uint32\_t* codeWidth = 8;

const *uint32\_t* lexemeWidth = 22;

*std*::*cout* << LIME << "============ mToken List: ============\n" << RESET;

*std*::*cout* << " Line" << " Pos" << " Code" << " Lexeme\n" << *std*::*endl*;

for (const Token& token : m\_LexerData->Tokens)

{

*int32\_t* padding = (lineWidth - *std*::*to\_string*(token.Line).*size*()) / 2;

*std*::*cout* << "|" << TEAL << *std*::*right* << *std*::*setw*(lineWidth - padding) << token.Line << RESET << *std*::*setw*(padding + 1) << "]";

padding = (posWidth - *std*::*to\_string*(token.Position).*size*()) / 2;

*std*::*cout*<< "[" << TEAL << *std*::*setw*(posWidth - padding) << token.Position << RESET << *std*::*setw*(padding+1) << "]";

padding = (codeWidth - *std*::*to\_string*(token.Code).*size*()) / 2;

*std*::*cout* << CRIMSON << *std*::*setw*(codeWidth - padding) << token.Code << RESET << *std*::*setw*(padding + 1) << "=";

*size\_t* lex = token.Lexeme.*size*();

padding = (lexemeWidth - lex) / 2;

*std*::*cout* << *std*::*setw*(lexemeWidth - lex - padding) << *std*::*right* << "<" << LEMON << token.Lexeme << RESET << ">" << *std*::*setw*(padding+1) << " " << *std*::*endl*;

}

*std*::*cout* << LIME << "=====================================\n\n" << RESET;

m\_Ofs << "============ Token List: ============\n";

m\_Ofs << " Line" << " Pos" << " Code" << " Lexeme\n" << *std*::*endl*;

for (const Token& token : m\_LexerData->Tokens)

{

*int32\_t* padding = (lineWidth - *std*::*to\_string*(token.Line).*size*()) / 2;

m\_Ofs << "|" << *std*::*right* << *std*::*setw*(lineWidth - padding) << token.Line << *std*::*setw*(padding + 1) << "]";

padding = (posWidth - *std*::*to\_string*(token.Position).*size*()) / 2;

m\_Ofs << "[" << *std*::*setw*(posWidth - padding) << token.Position << *std*::*setw*(padding + 1) << "]";

padding = (codeWidth - *std*::*to\_string*(token.Code).*size*()) / 2;

m\_Ofs << *std*::*setw*(codeWidth - padding) << token.Code << *std*::*setw*(padding + 1) << "=";

padding = (lexemeWidth - token.Lexeme.*size*()) / 2;

m\_Ofs << *std*::*setw*(lexemeWidth - padding) << *std*::*right* << "<" + token.Lexeme + ">" << *std*::*setw*(padding + 1) << " " << *std*::*endl*;

}

m\_Ofs << "=====================================\n\n";

}

void CLInterface::OutIdentifiersTable()

{

DisplayTable(m\_LexerData->IdentifiersTable, "Identifiers Table");

}

void CLInterface::OutConstantsTable()

{

DisplayTable(m\_LexerData->ConstantsTable, "Constants Table");

}

void CLInterface::OutKeywordsTable()

{

DisplayTable(m\_LexerData->KeyWordsTable, "Keywords Table");

}

void CLInterface::OutOptions()

{

*std*::*cout* << "Source file: \n";

*std*::*cout* << "Out file: \n";

}

void CLInterface::UsageHint(char\* name)

{

*std*::*cout* << "Usage: " << name << " <source\_file> [options...] <out\_file>\n";

}

void CLInterface::DisplayTable(const *std*::*unordered\_map*<*std*::*string*, *uint32\_t*>& table, const *std*::*string*& tableHeader)

{

const *uint32\_t* lexemeWidth = 25;

*std*::*cout* << "==========" << TEAL << tableHeader << ":" << RESET << "========= \n\n";

*std*::*cout* << "| Code" << "| Lexeme |" << *std*::*endl*;

*std*::*cout* << "+-----+-------------------------+" << *std*::*endl*;

for (auto& record : table)

{

*std*::*cout* << *std*::*left* << "|" << CRIMSON << *std*::*setw*(5) << record.*second* << RESET << "|";

*size\_t* lex = record.*first*.*size*();

*int32\_t* padding = (float)(lexemeWidth - lex) / 2.0;

*std*::*cout* << *std*::*setw*(lexemeWidth - lex - padding) << *std*::*right* << "<" << LEMON << record.*first* << RESET << ">" << *std*::*setw*(padding) << " |" << *std*::*endl*;

}

*std*::*cout* << "=====================================\n\n";

m\_Ofs << "========== " << tableHeader << ": =========\n\n";

m\_Ofs << "| Code" << "| Lexeme |" << *std*::*endl*;

m\_Ofs << "+-----+-------------------------+" << *std*::*endl*;

for (auto& record : table)

{

m\_Ofs << *std*::*left* << "|" << *std*::*setw*(5) << record.*second* << "|";

*int32\_t* padding = (float)(lexemeWidth - record.*first*.*size*()) / 2.0;

m\_Ofs << *std*::*setw*(lexemeWidth - padding) << *std*::*right* << "<" + record.*first* + ">" << *std*::*setw*(padding) << "|" << *std*::*endl*;

}

m\_Ofs << "=====================================\n\n";

}

*ErrorHandler.h*

#ifndef ERROR\_HANDLER\_H\_

#define ERROR\_HANDLER\_H\_

#include "Error.h"

#include <vector>

#include <string>

#include <memory>

class ErrorHandler

{

public:

ErrorHandler();

void ReportError(Error error);

*std*::*shared\_ptr*<*std*::*vector*<Error>> GetErrors();

static Error CreateSyntaxError(const *std*::*string*& errorMessage, *uint32\_t* line, *uint32\_t* pos, EErrorInstigator instigator);

static Error CreateGeneralError(const *std*::*string*& errorMessage, EErrorInstigator instigator);

void GotFatalError();

bool HasFatalError();

private:

static Error CreateError(const *std*::*string*& errorMessage, *uint32\_t* line, *uint32\_t* pos, EErrorInstigator instigator, EErrorType type);

private:

*std*::*shared\_ptr*<*std*::*vector*<Error>> m\_Errors;

bool bFatalError;

};

#endif /\* ERROR\_HANDLER\_H\_ \*/

*ErrorHandler.cpp*

#include "ErrorHandler.h"

#include <sstream>

#include <memory>

ErrorHandler::ErrorHandler() : bFatalError(false)

{

m\_Errors = *std*::*make\_shared*<*std*::*vector*<Error>>();

}

void ErrorHandler::ReportError(Error error)

{

m\_Errors->*push\_back*(error);

}

*std*::*shared\_ptr*<*std*::*vector*<Error>> ErrorHandler::GetErrors()

{

return m\_Errors;

}

Error ErrorHandler::CreateSyntaxError(const *std*::*string*& errorMessage, *uint32\_t* line, *uint32\_t* pos, EErrorInstigator instigator)

{

return CreateError(errorMessage, line, pos, instigator, EErrorType::SyntaxError);

}

Error ErrorHandler::CreateGeneralError(const *std*::*string*& errorMessage, EErrorInstigator instigator)

{

return CreateError(errorMessage, 0, 0, instigator, EErrorType::DriverError);

}

void ErrorHandler::GotFatalError()

{

bFatalError = true;

}

bool ErrorHandler::HasFatalError()

{

return bFatalError;

}

Error ErrorHandler::CreateError(const *std*::*string*& errorMessage, *uint32\_t* line, *uint32\_t* pos, EErrorInstigator instigator, EErrorType type)

{

return { errorMessage, line, pos, instigator, type };

}

*Error.h*

#ifndef ERROR\_H\_

#define ERROR\_H\_

#include <string>

#include <cstdint>

enum class EErrorInstigator : *uint8\_t*

{

FileIO,

Lexer,

Parser,

CodeGenerator

};

enum class EErrorType : *uint8\_t*

{

SyntaxError,

DriverError,

None

};

class Error

{

public:

Error(const *std*::*string*& msg, *size\_t* line, *size\_t* pos, EErrorInstigator inst, EErrorType type);

*std*::*string* GetMessage();

*std*::*string* GetInstigator();

*std*::*string* GetType();

*std*::*string* GetLine();

*std*::*string* GetPosition();

EErrorInstigator GetEnumInstigator();

private:

static *std*::*string* InstigatorToString(EErrorInstigator instigator);

static *std*::*string* TypeToString(EErrorType type);

protected:

*size\_t* m\_Line;

*size\_t* m\_Position;

EErrorInstigator m\_Instigator;

EErrorType m\_ErrorType;

*std*::*string* m\_Message;

};

#endif /\* ERROR\_H\_\*/

*Error.cpp*

#include "Error.h"

Error::Error(const *std*::*string*& msg, *size\_t* line, *size\_t* pos, EErrorInstigator inst, EErrorType type)

: m\_Message(msg), m\_Line(line), m\_Position(pos), m\_Instigator(inst), m\_ErrorType(type)

{

}

*std*::*string* Error::GetMessage()

{

return m\_Message;

}

*std*::*string* Error::GetInstigator()

{

return InstigatorToString(m\_Instigator);

}

*std*::*string* Error::GetType()

{

return TypeToString(m\_ErrorType);

}

*std*::*string* Error::GetLine()

{

return *std*::*to\_string*(m\_Line);

}

*std*::*string* Error::GetPosition()

{

return *std*::*to\_string*(m\_Position);

}

EErrorInstigator Error::GetEnumInstigator()

{

return m\_Instigator;

}

*std*::*string* Error::InstigatorToString(EErrorInstigator instigator)

{

switch (instigator)

{

case EErrorInstigator::FileIO:

return "File IO";

case EErrorInstigator::Lexer:

return "Lexer";

case EErrorInstigator::Parser:

return "Parser";

case EErrorInstigator::CodeGenerator:

return "Code Generator";

default:

return "Unknown";

}

}

*std*::*string* Error::TypeToString(EErrorType type)

{

switch (type)

{

case EErrorType::SyntaxError:

return "syntax error";

case EErrorType::DriverError:

return "driver error";

default:

return "unknown error";

}

}

*main.cpp*

#include "Control/Driver.h"

#include "Utilities/CompilerInterface.h"

#include "Utilities/Log.h"

/\*

\* 1. <signal-program> --> <program>

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

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

\* 4. <variable-declarations> --> VAR <declarations-list> | <empty>

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

\* 6. <declaration> --><variable-identifier>:<attribute> ;

\* 7. <attribute> --> INTEGER | FLOAT

\* 8. <statements-list> --> <statement> <statements-list> | <empty>

\* 9. <statement> --> <condition-statement> ENDIF ;

\* 10. <condition-statement> --> <incomplete-condition-statement><alternative-part>

\* 11. <incomplete-condition-statement> --> IF <conditional-expression> THEN <statements-list>

\* 12. <alternative-part> --> ELSE <statements-list> | <empty>

\* 13. <conditional-expression> --> <expression> = <expression>

\* 14. <expression> --> <variable-identifier> | <unsigned-integer>

\* 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

\*/

int main(int argc, char\* argv[])

{

Log::Init();

Driver driver;

driver.SetUI(*std*::*make\_unique*<CLInterface>());

driver.CreateOptionsFromCLArguments(argc, argv);

driver.Start();

return 0;

}

*Token.h*

#ifndef TOKEN\_H\_

#define TOKEN\_H\_

#include <string>

#include <cstdint>

enum class ETokenCode : *uint32\_t*

{

None = 0,

DelimiterBase = 301,

DelimiterAssign,

KW\_Base = 401,

KW\_PROGRAM,

KW\_VAR,

KW\_BEGIN,

KW\_END,

KW\_INTEGER,

KW\_FLOAT,

KW\_IF,

KW\_THEN,

KW\_ELSE,

KW\_ENDIF,

ConstantBase = 501,

IdentifierBase = 1001

};

struct Token

{

*size\_t* Line;

*size\_t* Position;

*uint32\_t* Code;

*std*::*string* Lexeme;

};

#endif /\* TOKEN\_H\_ \*/

*Lexer.h*

#ifndef LEXER\_H\_

#define LEXER\_H\_

#include "Token.h"

#include "Errors/Error.h"

#include <cstdint>

#include <fstream>

#include <array>

#include <vector>

#include <unordered\_map>

#include <memory>

enum class ESymbolCategories : *uint8\_t*

{

None = 0,

WhiteSpace,

Identifier,

Constant,

UnaryDelimiter,

MultiDelimiter,

Comment,

End

};

struct LexerData

{

*std*::*vector*<Token> Tokens;

*std*::*unordered\_map*<*std*::*string*, *uint32\_t*> ConstantsTable;

*std*::*unordered\_map*<*std*::*string*, *uint32\_t*> IdentifiersTable;

*std*::*unordered\_map*<*std*::*string*, *uint32\_t*> KeyWordsTable;

};

class ErrorHandler;

class Lexer

{

public:

Lexer(*std*::*shared\_ptr*<ErrorHandler> errorHandler);

void Scan(const *std*::*string*& filePath);

*std*::*shared\_ptr*<LexerData> GetLexerData();

private:

void Next();

void SetupSymbolCategories();

void SetupKeywordTable();

void WhiteSpaceState();

void IdentifierState();

void ConstantState();

void UnaryDelimiterState();

void MultiDelimiterState();

void CommentState();

void InCommentState(*size\_t* line, *size\_t* pos);

void EndCommentState(*size\_t* line, *size\_t* pos);

private:

*std*::*ifstream* m\_InputFile;

// Lexer state

*size\_t* m\_Line;

*size\_t* m\_Position;

char m\_CurrentCharacter;

ESymbolCategories m\_CurrentSymbol;

*std*::array<ESymbolCategories, 128> m\_Attributes;

// Token-related

*std*::*string* m\_TokenBuffer;

*std*::*vector*<Token> m\_Tokens;

// Tables

*std*::*unordered\_map*<*std*::*string*, *uint32\_t*> m\_ConstantsTable;

*std*::*unordered\_map*<*std*::*string*, *uint32\_t*> m\_IdentifiersTable;

*std*::*unordered\_map*<*std*::*string*, *uint32\_t*> m\_KeyWordsTable;

*std*::*shared\_ptr*<ErrorHandler> m\_ErrorHandler;

EErrorInstigator m\_Instigator;

};

#endif /\* LEXER\_H\_ \*/

*Lexer.cpp*

#include "Lexer.h"

#include "Errors/ErrorHandler.h"

Lexer::Lexer(*std*::*shared\_ptr*<ErrorHandler> errorHandler)

: m\_Line(1), m\_Position(0), m\_CurrentCharacter(0), m\_ErrorHandler(errorHandler), m\_Instigator(EErrorInstigator::Lexer)

{

SetupSymbolCategories();

SetupKeywordTable();

}

void Lexer::Scan(const *std*::*string*& filePath)

{

m\_InputFile.*open*(filePath);

if (!m\_InputFile.*is\_open*())

{

auto error = ErrorHandler::CreateGeneralError(*std*::*string*("No such file or directory: ") + filePath, EErrorInstigator::FileIO);

m\_ErrorHandler->ReportError(error);

m\_ErrorHandler->GotFatalError();

return;

}

Next();

while (m\_CurrentSymbol != ESymbolCategories::End)

{

switch (m\_CurrentSymbol)

{

case ESymbolCategories::WhiteSpace:

WhiteSpaceState();

break;

case ESymbolCategories::Identifier:

IdentifierState();

break;

case ESymbolCategories::Constant:

ConstantState();

break;

case ESymbolCategories::UnaryDelimiter:

UnaryDelimiterState();

break;

case ESymbolCategories::MultiDelimiter:

MultiDelimiterState();

break;

case ESymbolCategories::Comment:

CommentState();

break;

default:

auto error = ErrorHandler::CreateSyntaxError(*std*::*string*("Illegal character '") + m\_CurrentCharacter + "' found", m\_Line, m\_Position, m\_Instigator);

m\_ErrorHandler->ReportError(error);

Next();

break;

}

}

m\_InputFile.*close*();

}

*std*::*shared\_ptr*<LexerData> Lexer::GetLexerData()

{

return *std*::*make\_shared*<LexerData>( m\_Tokens, m\_ConstantsTable, m\_IdentifiersTable, m\_KeyWordsTable);

}

void Lexer::Next()

{

m\_CurrentCharacter = m\_InputFile.*get*();

if (m\_CurrentCharacter == '\n')

{

m\_Line++;

m\_Position = 0;

}

else if (m\_CurrentCharacter == '\t')

{

m\_Position += 4;

}

else

{

m\_Position++;

}

if (m\_CurrentCharacter != *EOF*)

{

m\_CurrentSymbol = m\_Attributes[m\_CurrentCharacter];

}

else

{

m\_CurrentSymbol = ESymbolCategories::End;

}

}

void Lexer::SetupSymbolCategories()

{

m\_Attributes.*fill*(ESymbolCategories::None);

for (*size\_t* i = 8; i < 14; i++)

m\_Attributes[i] = ESymbolCategories::WhiteSpace;

m\_Attributes[32] = ESymbolCategories::WhiteSpace;

for (*size\_t* i = 48; i < 58; i++)

m\_Attributes[i] = ESymbolCategories::Constant;

for (*size\_t* i = 64; i < 91; i++)

m\_Attributes[i] = ESymbolCategories::Identifier; // [@-A-Z]

for (*size\_t* i = 97; i < 123; i++)

m\_Attributes[i] = ESymbolCategories::Identifier; // [a-z]

m\_Attributes['\_'] = ESymbolCategories::Identifier;

m\_Attributes['('] = ESymbolCategories::Comment;

m\_Attributes[':'] = ESymbolCategories::MultiDelimiter;

m\_Attributes[';'] = ESymbolCategories::UnaryDelimiter;

m\_Attributes['='] = ESymbolCategories::UnaryDelimiter;

m\_Attributes['.'] = ESymbolCategories::UnaryDelimiter;

}

void Lexer::SetupKeywordTable()

{

m\_KeyWordsTable["PROGRAM"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_PROGRAM);

m\_KeyWordsTable["VAR"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_VAR);

m\_KeyWordsTable["BEGIN"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_BEGIN);

m\_KeyWordsTable["END"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_END);

m\_KeyWordsTable["INTEGER"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_INTEGER);

m\_KeyWordsTable["FLOAT"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_FLOAT);

m\_KeyWordsTable["IF"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_IF);

m\_KeyWordsTable["THEN"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_THEN);

m\_KeyWordsTable["ELSE"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_ELSE);

m\_KeyWordsTable["ENDIF"] = static\_cast<*uint32\_t*>(ETokenCode::KW\_ENDIF);

m\_KeyWordsTable[":="] = static\_cast<*uint32\_t*>(ETokenCode::DelimiterAssign);

}

void Lexer::WhiteSpaceState()

{

while (m\_CurrentSymbol == ESymbolCategories::WhiteSpace)

{

Next();

}

}

void Lexer::IdentifierState()

{

*size\_t* lexemeLine = m\_Line;

*size\_t* lexemeStartPosition = m\_Position;

while (m\_CurrentSymbol == ESymbolCategories::Identifier ||

m\_CurrentSymbol == ESymbolCategories::Constant)

{

m\_TokenBuffer += m\_CurrentCharacter;

Next();

}

*uint32\_t* lexemeCode;

if (auto kwRecord = m\_KeyWordsTable.*find*(m\_TokenBuffer); kwRecord != m\_KeyWordsTable.*end*())

{

lexemeCode = kwRecord->*second*;

}

else if (auto identifierRecord = m\_IdentifiersTable.*find*(m\_TokenBuffer); identifierRecord == m\_IdentifiersTable.*end*())

{

lexemeCode = static\_cast<*uint32\_t*>(ETokenCode::IdentifierBase) + m\_IdentifiersTable.*size*();

m\_IdentifiersTable[m\_TokenBuffer] = lexemeCode;

}

else

{

lexemeCode = identifierRecord->*second*;

}

m\_Tokens.*emplace\_back*(lexemeLine, lexemeStartPosition, lexemeCode, m\_TokenBuffer);

m\_TokenBuffer.*clear*();

}

void Lexer::ConstantState()

{

*size\_t* lexemeLine = m\_Line;

*size\_t* lexemeStartPosition = m\_Position;

while (m\_CurrentSymbol == ESymbolCategories::Constant)

{

m\_TokenBuffer += m\_CurrentCharacter;

Next();

}

*uint32\_t* lexemeCode;

if (auto tableRecord = m\_ConstantsTable.*find*(m\_TokenBuffer); tableRecord == m\_ConstantsTable.*end*())

{

lexemeCode = static\_cast<*uint32\_t*>(ETokenCode::ConstantBase) + m\_ConstantsTable.*size*();

m\_ConstantsTable[m\_TokenBuffer] = lexemeCode;

}

else

{

lexemeCode = tableRecord->*second*;

}

m\_Tokens.*emplace\_back*(lexemeLine, lexemeStartPosition, lexemeCode, m\_TokenBuffer);

m\_TokenBuffer.*clear*();

}

void Lexer::UnaryDelimiterState()

{

*size\_t* lexemeLine = m\_Line;

*size\_t* lexemeStartPosition = m\_Position;

m\_TokenBuffer += m\_CurrentCharacter;

*uint32\_t* lexemeCode = static\_cast<*uint32\_t*>(m\_CurrentSymbol);

Next();

m\_Tokens.*emplace\_back*(lexemeLine, lexemeStartPosition, lexemeCode, m\_TokenBuffer);

m\_TokenBuffer.*clear*();

}

void Lexer::MultiDelimiterState()

{

*size\_t* lexemeLine = m\_Line;

*size\_t* lexemeStartPosition = m\_Position;

m\_TokenBuffer += m\_CurrentCharacter;

*uint32\_t* lexemeCode = static\_cast<*uint32\_t*>(m\_CurrentSymbol);

Next();

if (m\_CurrentCharacter == '=')

{

m\_TokenBuffer += m\_CurrentCharacter;

if (auto tableRecord = m\_KeyWordsTable.*find*(m\_TokenBuffer); tableRecord != m\_KeyWordsTable.*end*())

{

lexemeCode = tableRecord->*second*;

}

}

m\_Tokens.*emplace\_back*(lexemeLine, lexemeStartPosition, lexemeCode, m\_TokenBuffer);

m\_TokenBuffer.*clear*();

}

void Lexer::CommentState()

{

*size\_t* comStartLine = m\_Line;

*size\_t* comStartPos = m\_Position;

Next();

if (m\_CurrentCharacter == '\*')

{

InCommentState(comStartLine, comStartPos);

}

else

{

auto error = ErrorHandler::CreateSyntaxError("Missing '\*' after '(' in comment", comStartLine, comStartPos, m\_Instigator);

m\_ErrorHandler->ReportError(error);

Next();

}

}

void Lexer::InCommentState(*size\_t* line, *size\_t* pos)

{

Next();

while (m\_CurrentCharacter != '\*' && m\_CurrentCharacter != *EOF*)

{

Next();

}

if (m\_CurrentCharacter == '\*')

{

EndCommentState(line, pos);

}

else

{

auto error = ErrorHandler::CreateSyntaxError("Comment not closed", line, pos, m\_Instigator);

m\_ErrorHandler->ReportError(error);

}

}

void Lexer::EndCommentState(*size\_t* line, *size\_t* pos)

{

while (m\_CurrentCharacter == '\*')

{

Next();

}

if (m\_CurrentCharacter == ')')

{

Next();

return;

}

InCommentState(line, pos);

}

*Driver.h*

#ifndef DRIVER\_H\_

#define DRIVER\_H\_

#include "Compiler.h"

#include "Utilities/CompilerInterface.h"

#include <string>

#include <memory>

class ErrorHandler;

struct Options

{

*std*::*string* SourceFile;

*std*::*string* OutputFile;

};

class Driver final

{

public:

Driver();

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

void Start();

void SetUI(*std*::*unique\_ptr*<CompilerInterface>&& ui);

private:

bool CheckSourceExtension(const *std*::*string*& filePath);

void Terminate();

private:

Options m\_Options;

*std*::*unique\_ptr*<Compiler> m\_Compiler;

*std*::*shared\_ptr*<ErrorHandler> m\_ErrorHandler;

*std*::*unique\_ptr*<CompilerInterface> m\_UI;

};

#endif /\* DRIVER\_H\_ \*/

*Driver.cpp*

#include "Driver.h"

#include "Errors/ErrorHandler.h"

#include "Errors/Error.h"

#include "Lexer/Lexer.h"

#include "Utilities/Log.h"

#include <stdlib.h>

Driver::Driver()

{

m\_ErrorHandler = *std*::*make\_shared*<ErrorHandler>();

m\_Compiler = *std*::*make\_unique*<Compiler>(m\_ErrorHandler);

}

void Driver::CreateOptionsFromCLArguments(int argc, char\* argv[])

{

if (argc < 2)

{

m\_UI->UsageHint(argv[0]);

Terminate();

return;

}

bool matches = CheckSourceExtension(argv[1]);

if (!matches)

{

m\_UI->OutErrors();

Terminate();

return;

}

m\_Options.SourceFile = argv[1];

for (int i = 2; i < argc-1; i++)

{

if (*std*::*string*(argv[i]) == "-o")

{

m\_Options.OutputFile = argv[i + 1];

}

}

}

bool Driver::CheckSourceExtension(const *std*::*string*& filePath)

{

*size\_t* dotPos = filePath.*find*('.');

if (dotPos != *std*::*string*::*npos*)

{

*std*::*string* extension = filePath.*substr*(dotPos + 1);

if (extension != "sig")

{

auto error = ErrorHandler::CreateGeneralError(*std*::*string*("Incorrect input file type: .") + extension + "\n message: expected \".sig\"", EErrorInstigator::FileIO);

m\_ErrorHandler->ReportError(error);

return false;

}

}

return true;

}

void Driver::Terminate()

{

LOG\_ERROR("Compilation terminated");

*exit*(*EXIT\_FAILURE*);

}

void Driver::Start()

{

LOG\_STATE("Build started...", "\n");

m\_UI->SetInfoFileName("comp\_info.txt");

m\_UI->SetOutToFileEnabled(true);

m\_Compiler->Compile(m\_Options.SourceFile);

m\_UI->SetLexerData(m\_Compiler->GetLexerData());

m\_UI->OutErrors();

if (m\_ErrorHandler->HasFatalError())

{

Terminate();

return;

}

m\_UI->OutTokens();

m\_UI->OutIdentifiersTable();

m\_UI->OutConstantsTable();

m\_UI->OutKeywordsTable();

}

void Driver::SetUI(*std*::*unique\_ptr*<CompilerInterface>&& ui)

{

m\_UI = *std*::*move*(ui);

m\_UI->SetErrorHandler(m\_ErrorHandler);

}

*Compiler.h*

#ifndef COMPILER\_H\_

#define COMPILER\_H\_

#include "Lexer/Lexer.h"

#include <string>

class ErrorHandler;

class Compiler

{

public:

Compiler(*std*::*shared\_ptr*<ErrorHandler> errorHandler);

void Compile(const *std*::*string*& filePath);

*std*::*shared\_ptr*<LexerData> GetLexerData();

private:

private:

*std*::*unique\_ptr*<Lexer> m\_Lexer; // May be store in shared\_ptr and create in driver

*std*::*shared\_ptr*<ErrorHandler> m\_ErrorHandler;

};

#endif /\* COMPILER\_H\_ \*/

*Compiler.cpp*

#include "Compiler.h"

#include "Errors/ErrorHandler.h"

Compiler::Compiler(*std*::*shared\_ptr*<ErrorHandler> errorHandler) : m\_ErrorHandler(errorHandler)

{

m\_Lexer = *std*::*make\_unique*<Lexer>(errorHandler);

}

void Compiler::Compile(const *std*::*string*& filePath)

{

m\_Lexer->Scan(filePath);

}

*std*::*shared\_ptr*<LexerData> Compiler::GetLexerData()

{

return m\_Lexer->GetLexerData();

}

**Контрольні приклади**