

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО"

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

**Лабораторна робота 1**

з дисципліни “Теорія формальних мов та компіляція”

|  |  |  |
| --- | --- | --- |
| Виконав  студент 5 курсу  групи КП- 91-мн  Іващенко Михайло  варіант № 20 |  |  |

Київ 2019

**ГРАМАТИКА**

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

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

3. <block> --> <variable-declarations> BEGIN

<statements-list> END

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

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

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

7. <attribute> --> INTEGER | FLOAT

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

9. <statement> --> WHILE <conditional-expression> DO <statements-list> ENDWHILE ;

10. <conditional-expression> --> <expression>

<comparison-operator><expression>  
11. <comparison-operator> --> < |

<= |

= |

<> |

>= |

>

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

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

14. <procedure-identifier> --> <identifier>

15. <identifier> --> <letter><string>

16. <string> --> <letter><string> |

<digit><string> |

<empty>

17. <unsigned-integer> --> <digit><digits-string>

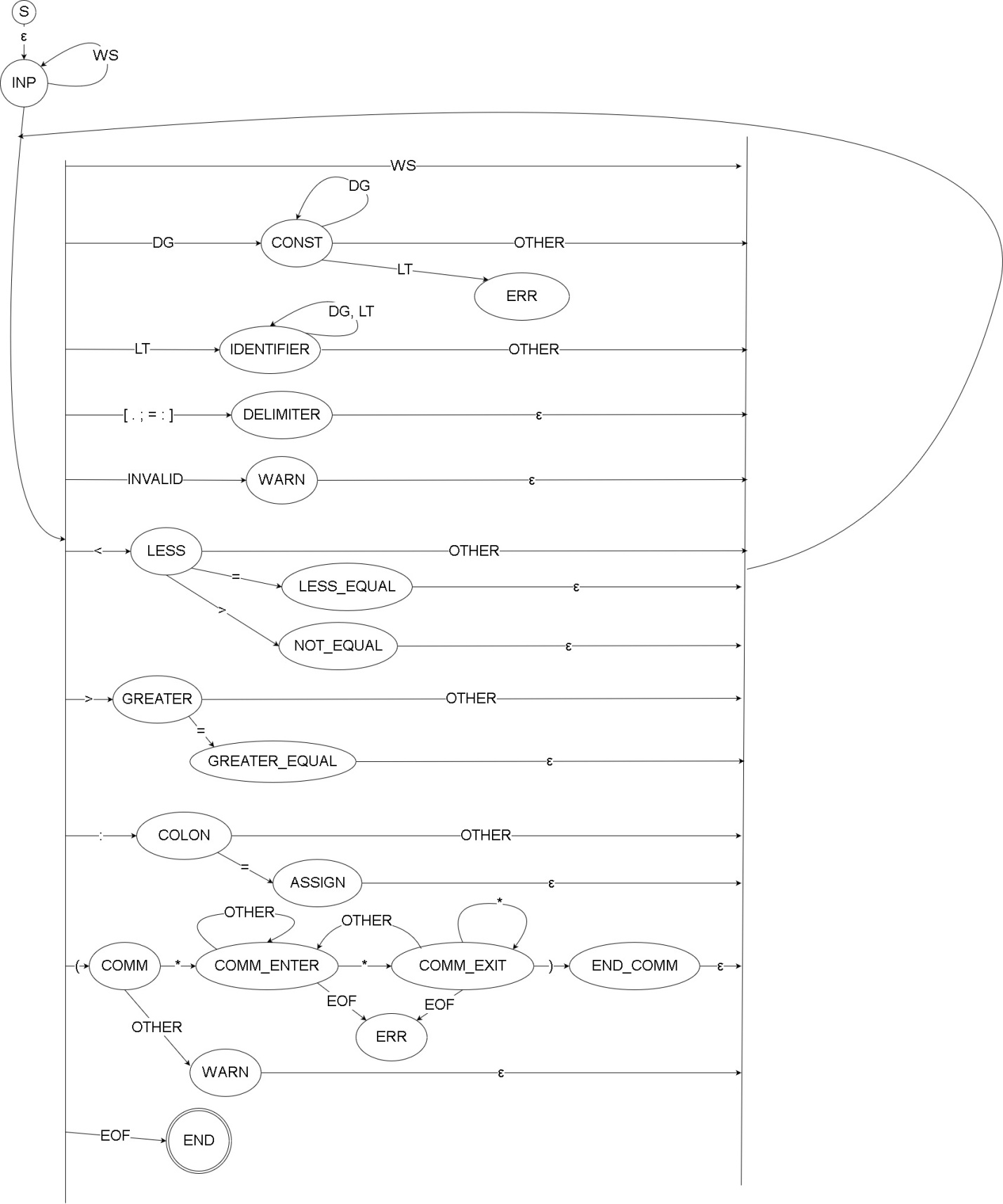
18. <digits-string> --> <digit><digits-string> | <empty>

19. <digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

9

20. <letter> --> A | B | C | D | ... | Z

Граф автомату



Код програми

**LookupTable.h**

**#ifndef** LOOKUPTABLE\_H\_

**#define** LOOKUPTABLE\_H\_

**#include** <string>

**#include** <unordered\_map>

**#include** <fstream>

**#include** <vector>

**enum** SearchResult

{

*notFound*, *found*, *midWayFound*, *midWayNotFound*

};

**enum** CharacterClass

{

*invalid* = 256,

*simpleDelimiter* = 1,

*complexDelimiterStart* = 2,

*letter* = 4,

*whitespace* = 8,

*constantStart* = 16,

*identifierStart* = 32,

*commentStart* = 64,

*digit* = 128,

*complexDelimiter* = 512

};

**template** <**typename** **LexemType**, **typename** **CodeType**>

**class** InformationTable

{

**public**:

**InformationTable**(){};

**virtual** **~InformationTable**(){};

**struct** Search

{

**public**:

**int** found;

**CodeType** code;

**Search**(**CodeType** \_code, **int** \_found = 0)

{

code = \_code;

found = \_found;

}

};

**virtual** Search **find**(**LexemType** lexem) = 0;

**virtual** **void** **read**(std::string filename, **char** delimiter) = 0;

**virtual** **void** **add**(**LexemType** lex, **CodeType** cod) = 0;

**virtual** **void** **put**(std::ostream& steam) = 0;

**virtual** **int** **getSize**() = 0;

};

**#endif** /\* LOOKUPTABLE\_H\_ \*/

**AttributeTable.h**

**#ifndef** ATTRIBUTETABLE\_H\_

**#define** ATTRIBUTETABLE\_H\_

**#include** "LookupTable.h"

**class** AttributeTable : **public** InformationTable<**char**, **int**>

{

**private**:

std::unordered\_map<**char**, **int**> information;

**const** **int** defaultCap = 127;

**public**:

**AttributeTable**();

**~AttributeTable**();

InformationTable<**char**, **int**>::Search **find**(**char** lexem);

**void** **read**(std::string path, **char** delimiter = ' ');

**void** **add**(**char** lex, **int** cod);

**void** **put**(std::ostream& stream);

**int** **getSize**();

};

**#endif** /\* ATTRIBUTETABLE\_H\_ \*/

**AttributeTable.cpp**

**#include** "AttributeTable.h"

**AttributeTable::AttributeTable**() : InformationTable<**char**, **int**>()

{

information = std::unordered\_map<**char**, **int**>(defaultCap);

}

**AttributeTable::~AttributeTable**()

{

}

InformationTable<**char**, **int**>::Search **AttributeTable::find**(**char** lexem)

{

Search res(0);

**auto** search = information.find(lexem);

**if**(search != information.end())

{

res.found = SearchResult::*found*;

res.code = search->second;

}

**else**

{

res.found = SearchResult::*notFound*;

res.code = -1;

}

**return** res;

}

**void** **AttributeTable::read**(std::string path, **char** delimiter)

{

std::ifstream input(path);

std::string str;

std::string identifier;

**int** code;

**char** character;

**while**(std::getline(input, str))

{

**int** delimiterPosition = str.rfind(delimiter);

identifier = str.substr(0, delimiterPosition);

code = **atoi**(str.substr(delimiterPosition+1).c\_str());

character = **static\_cast**<**char**>(**atoi**(identifier.c\_str()));

information.insert({character, code});

}

input.close();

}

**void** **AttributeTable::add**(**char** lex, **int** cod)

{

information.insert({lex, cod});

}

**void** **AttributeTable::put**(std::ostream& stream)

{

**for**(**auto**& p : information)

{

stream << p.first << " " << p.second << std::**endl**;

}

}

**int** **AttributeTable::getSize**()

{

**return** information.size();

}

**#ifndef** LEXEMTABLE\_H\_

**#define** LEXEMTABLE\_H\_

**#include** "LookupTable.h"

**LexemTable.h**

**class** LexemTable : **public** InformationTable<std::string, **int**>

{

**private**:

std::unordered\_map<std::string, **int**> information;

**const** **int** defaultCap = 127;

**public**:

**LexemTable**();

**~LexemTable**();

InformationTable<std::string, **int**>::Search **find**(std::string lexem);

**void** **read**(std::string path, **char** delimiter = ' ');

**void** **add**(std::string lex, **int** cod);

**void** **put**(std::ostream& stream);

**int** **getSize**();

};

**#endif** /\* LEXEMTABLE\_H\_ \*/

**LexemTable.cpp**

**#include** "LexemTable.h"

**LexemTable::LexemTable**() : InformationTable<std::string, **int**>()

{

information = std::unordered\_map<std::string, **int**>(defaultCap);

}

**LexemTable::~LexemTable**()

{

}

InformationTable<std::string, **int**>::Search **LexemTable::find**(std::string lexem)

{

Search res(0);

**auto** search = information.find(lexem);

**if**(search != information.end())

{

res.found = SearchResult::*found*;

res.code = search->second;

}

**else**

{

res.found = SearchResult::*notFound*;

res.code = -1;

}

**return** res;

}

**void** **LexemTable::read**(std::string path, **char** delimiter)

{

std::ifstream input(path);

std::string str;

std::string identifier;

**int** code;

**while**(std::getline(input, str))

{

**int** delimiterPosition = str.rfind(delimiter);

identifier = str.substr(0, delimiterPosition);

code = **atoi**(str.substr(delimiterPosition+1).c\_str());

information.insert({identifier, code});

}

input.close();

}

**void** **LexemTable::add**(std::string lex, **int** cod)

{

information.insert({lex, cod});

}

**void** **LexemTable::put**(std::ostream& stream)

{

**for**(**auto**& p: information)

{

stream << p.first << " " << p.second << std::**endl**;

}

}

**int** **LexemTable::getSize**()

{

**return** information.size();

}

**PrefixTree.h**

**#ifndef** PREFIXTREE\_H\_

**#define** PREFIXTREE\_H\_

**#include** "LookupTable.h"

**#include** <vector>

**class** PrefixTree : **public** InformationTable<std::string, **int**>

{

**private**:

**int** size;

**public**:

**class** Node

{

**public**:

**char** value;

**int** code;

std::vector<Node\*> children;

**Node**(**char** c, **int** \_code = -1);

**~Node**();

**int** **find**(**char** next);

**void** **add**(**char** newC);

};

Node\* root;

**PrefixTree**();

InformationTable<std::string, **int**>::Search **find**(std::string lexem);

**void** **read**(std::string path, **char** delimiter = ' ');

**void** **add**(std::string lex, **int** cod);

**void** **put**(std::ostream& stream);

**int** **getSize**();

**~PrefixTree**();

**private**:

**void** **insert**(std::string del, **int** code);

**void** **printNode**(Node\* n, std::ostream& stream, std::string formed = "");

};

**#endif** /\* PREFIXTREE\_H\_ \*/

**PrefixTree.cpp**

**#include** "PrefixTree.h"

**PrefixTree::Node::Node**(**char** c, **int** \_code)

{

code = \_code;

value = c;

children = std::vector<Node\*>();

}

**PrefixTree::Node::~Node**()

{

**for**(**unsigned** **int** i = 0; i < children.size(); i++)

{

**delete** children[i];

}

children.clear();

}

**int** **PrefixTree::Node::find**(**char** next)

{

**unsigned** **int** i = 0;

**for**(; i < children.size(); i++)

{

**if**(children[i]->value == next)

{

**return** i;

}

}

**return** -1;

}

**void** **PrefixTree::Node::add**(**char** newC)

{

children.push\_back(**new** Node(newC));

}

**PrefixTree::PrefixTree**() : InformationTable()

{

root = **new** Node(0, -1);

size = 0;

}

InformationTable<std::string, **int**>::Search **PrefixTree::find**(std::string lexem)

{

Node\* current = root;

**unsigned** **int** i = 0;

Search res(SearchResult::*notFound*);

**for**(i = 0; i < lexem.length(); i++)

{

**int** pos = current->find(lexem[i]);

**if**(pos != -1)

{

current = current->children[pos];

}

**else**

{

res.found = SearchResult::*midWayNotFound*;

res.code = current->code;

**break**;

}

}

**if**(i == lexem.length())

{

**if**(current->code == -1)

{

res.found = SearchResult::*midWayNotFound*;

res.code = -1;

}

**else**

{

res.code = current->code;

**if**(current->children.size() == 0)

{

res.found = SearchResult::*found*;

}

**else**

{

res.found = SearchResult::*midWayFound*;

}

}

}

**else**

{

res.found = SearchResult::*notFound*;

res.code = -1;

}

**return** res;

}

**void** **PrefixTree::read**(std::string path, **char** delimiter)

{

std::ifstream input(path);

std::string str;

std::string identifier;

**int** code;

**while**(std::getline(input, str))

{

**int** delimiterPosition = str.rfind(delimiter);

identifier = str.substr(0, delimiterPosition);

code = **atoi**(str.substr(delimiterPosition+1).c\_str());

insert(identifier, code);

}

input.close();

}

**void** **PrefixTree::add**(std::string lex, **int** cod)

{

insert(lex, cod);

}

**void** **PrefixTree::put**(std::ostream& stream)

{

printNode(root, stream, "");

}

**int** **PrefixTree::getSize**()

{

**return** size;

}

**PrefixTree::~PrefixTree**()

{

**delete** root;

}

**void** **PrefixTree::insert**(std::string del, **int** code)

{

Node\* currentNode = root;

**int** currentChar = 0;

**int** pos;

**while**((pos = currentNode->find(del[currentChar])) != -1)

{

currentChar++;

currentNode = currentNode->children[pos];

}

**for**(**unsigned** **int** i = currentChar; i < del.length(); i++)

{

currentNode->add(del[i]);

currentNode = currentNode->children[currentNode->find(del[i])];

}

currentNode->code = code;

size++;

}

**void** **PrefixTree::printNode**(Node\* n, std::ostream& stream, std::string formed)

{

**if**(n->children.size() > 0)

{

**for**(**unsigned** **int** i = 0; i < n->children.size(); i++)

{

printNode(n->children[i], stream, formed + n->value);

}

}

**else**

{

stream << formed << " " << n->code << std::**endl**;

}

}

**Configuration.h**

**#ifndef** CONFIGURATION\_H\_

**#define** CONFIGURATION\_H\_

**#include** <string>

**struct** Configuration

{

**public**:

**enum** TableID

{

*keywords*,

*identifiers*,

*delimiters*,

*constants*,

*attributes*,

*TABLE\_COUNT*

};

**Configuration**()

{

tables = **new** std::string[TableID::*TABLE\_COUNT*];

constantDiapasonStart = 100001;

identifierDiapasonStart = 601;

commentStart = -1;

commentEnd = -1;

}

std::string\* tables;

**int** constantDiapasonStart;

**int** identifierDiapasonStart;

**int** commentStart;

**int** commentEnd;

**~Configuration**()

{

**delete** tables;

//tables = 0;

}

};

**#endif** /\* CONFIGURATION\_H\_ \*/

**Lexem.h**

**#ifndef** LEXEM\_H\_

**#define** LEXEM\_H\_

**#include** <string>

**struct** EncodedLexem

{

**public**:

std::string lexem;

**int** code;

**int** row;

**int** column;

**EncodedLexem**(std::string initLexem = "", **int** initCode = 0, **int** initRow = 0, **int** initColumn = 0)

{

lexem = initLexem;

code = initCode;

row = initRow;

column = initColumn;

}

};

**#endif** /\* LEXEM\_H\_ \*/

**Lexer.h**

**#ifndef** LEXER\_H\_

**#define** LEXER\_H\_

**#include** <string>

**#include** <iostream>

**#include** "LookupTable.h"

**#include** "Configuration.h"

**#include** "Lexem.h"

**#include** "AttributeTable.h"

**#include** "LexemTable.h"

**#include** "PrefixTree.h"

**#define** IS(STATE, FLAG) ((STATE & FLAG) == FLAG)

**typedef** std::vector<EncodedLexem> LexemString;

**class** Lexer

{

**private**:

AttributeTable\* attributes;

InformationTable<std::string, **int**>\*\* tables;

Configuration config;

**int** row;

**int** col;

**int** startingRow;

**int** startingCol;

std::string errorMessage;

**public**:

**enum** StateFlag

{

*none* = 1, *error*, *constant*, *identifier*, *comment*, *delimiter*, *criticalError*

};

**Lexer**(Configuration c);

**~Lexer**();

InformationTable<std::string, **int**>\* **getTable**(Configuration::TableID id);

LexemString **processSource**(std::ifstream& codeStream);

**private**:

std::string preservedLexem;

StateFlag **processLexem**(std::string& lexem, LexemString& buffer, StateFlag expected);

};

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

**Lexer.cpp**

**#include** "Lexer.h"

Lexer::Lexer(Configuration c)

{

config = c;

attributes = **new** AttributeTable();

attributes->read(config.tables[Configuration::TableID::attributes], ' ');

tables=**new** InformationTable<std::string, **int**>\*[config.TableID::TABLE\_COUNT-1];

**for**(**int** i = 0; i < Configuration::TableID::TABLE\_COUNT-1; i++)

{

**if**(i != Configuration::TableID::delimiters)

{

tables[i] = **new** LexemTable();

}

**else**

{

tables[i] = **new** PrefixTree();

}

tables[i]->read(config.tables[i], ' ');

}

row = 1;

col = 0;

startingRow = 1;

startingCol = 0;

errorMessage = "";

}

Lexer::~Lexer()

{

**delete** attributes;

**for**(**int** i = 0; i < config.TableID::TABLE\_COUNT-1; i++)

{

**delete** tables[i];

}

**delete**[] tables;

}

InformationTable<std::string, **int**>\* Lexer::getTable(Configuration::TableID id)

{

**return** tables[id];

}

LexemString Lexer::processSource(std::ifstream& codeStream)

{

LexemString result;

StateFlag state = StateFlag::none;

**char** currentChar;

std::string storedLexem = "";

**while**(codeStream.get(currentChar))

{

**if**(currentChar == '\n')

{

row++;

col = 0;

}

**else**

{

col++;

}

CharacterClass currentClass;

**auto** search = attributes->find(currentChar);

**if**(search.found != SearchResult::notFound)

{

currentClass = (CharacterClass)(search.code);

}

**else**

{

currentClass = CharacterClass::invalid;

}

**if**(state == StateFlag::criticalError)

{

**break**;

}

**else**

{

**switch**(state)

{

**case** StateFlag::none:

startingCol = col;

startingRow = row;

**if**(IS(currentClass, CharacterClass::invalid))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiterStart))

{

storedLexem = "";

storedLexem += currentChar;

state = processLexem(storedLexem, result, StateFlag::delimiter);

}

**else** **if**(IS(currentClass, CharacterClass::simpleDelimiter))

{

storedLexem = "";

storedLexem += currentChar;

state = processLexem(storedLexem, result, StateFlag::none);

storedLexem = "";

}

**else** **if**(IS(currentClass, CharacterClass::constantStart))

{

storedLexem += currentChar;

state = StateFlag::constant;

}

**else** **if**(IS(currentClass, CharacterClass::identifierStart))

{

storedLexem += currentChar;

state = StateFlag::identifier;

}

**else** **if**(IS(currentClass, CharacterClass::whitespace))

{

**break**;

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiter))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid use of character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**break**;

**case** StateFlag::identifier:

**if**(IS(currentClass, CharacterClass::invalid))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**else** **if**(IS(currentClass, CharacterClass::letter) || IS(currentClass, CharacterClass::digit))

{

storedLexem += currentChar;

}

**else** **if**(IS(currentClass, CharacterClass::whitespace))

{

state = processLexem(storedLexem, result, StateFlag::identifier);

storedLexem = "";

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiterStart))

{

processLexem(storedLexem, result, StateFlag::identifier);

startingCol = col;

startingRow = row;

storedLexem = "";

storedLexem += currentChar;

state = StateFlag::delimiter;

}

**else** **if**(IS(currentClass, CharacterClass::simpleDelimiter))

{

processLexem(storedLexem, result, StateFlag::identifier);

startingCol = col;

startingRow = row;

storedLexem = "";

storedLexem += currentChar;

state = processLexem(storedLexem, result, StateFlag::none);

storedLexem = "";

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiter))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid use of character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**break**;

**case** StateFlag::constant:

**if**(IS(currentClass, CharacterClass::invalid))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**else** **if**(IS(currentClass, CharacterClass::digit))

{

storedLexem += currentChar;

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiterStart))

{

processLexem(storedLexem, result, StateFlag::constant);

startingCol = col;

startingRow = row;

storedLexem = "";

storedLexem += currentChar;

state = StateFlag::delimiter;

}

**else** **if**(IS(currentClass, CharacterClass::simpleDelimiter))

{

processLexem(storedLexem, result, StateFlag::constant);

startingCol = col;

startingRow = row;

storedLexem = "";

storedLexem += currentChar;

state = processLexem(storedLexem, result, StateFlag::none);

storedLexem = "";

}

**else** **if**(IS(currentClass, CharacterClass::whitespace))

{

state = processLexem(storedLexem, result, StateFlag::constant);

storedLexem = "";

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiter))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid use of character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**else**

{

**char** cMessage[250];

sprintf(cMessage, "Error: invalid constant format at %i; %i", row, col);

errorMessage = std::string(cMessage);

state = StateFlag::criticalError;

}

**break**;

**case** StateFlag::comment:

**if**(IS(currentClass, CharacterClass::complexDelimiterStart) || IS(currentClass, CharacterClass::simpleDelimiter))

{

storedLexem += currentChar;

**if**(storedLexem.length() > 1)

{

**auto** f = tables[Configuration::TableID::delimiters]->find(storedLexem);

**if**((f.found == SearchResult::found || f.found == SearchResult::midWayFound ) && f.code == config.commentEnd)

{

state = StateFlag::none;

storedLexem = "";

}

**else**

{

storedLexem = storedLexem.substr(1);

}

}

}

**else**

{

storedLexem = "";

}

**break**;

**case** StateFlag::delimiter:

**if**(IS(currentClass, CharacterClass::invalid))

{

**char** cMessage[250];

sprintf(cMessage, "Warning: ignoring invalid character %c at %i; %i", currentChar, row, col);

errorMessage = std::string(cMessage);

std::cerr << errorMessage << std::endl;

}

**else** **if**(IS(currentClass, CharacterClass::whitespace))

{

state = processLexem(storedLexem, result, StateFlag::none);

storedLexem = "";

}

**else** **if**(IS(currentClass, CharacterClass::complexDelimiter) || IS(currentClass, CharacterClass::complexDelimiterStart))

{

storedLexem += currentChar;

state = processLexem(storedLexem, result, StateFlag::delimiter);

}

**else** **if**(IS(currentClass, CharacterClass::constantStart))

{

state = processLexem(storedLexem, result, StateFlag::none);

**if**(state != StateFlag::error && state != StateFlag::criticalError)

{

state = StateFlag::constant;

startingCol = col;

startingRow = row;

storedLexem = "";

storedLexem += currentChar;

}

}

**else** **if**(IS(currentClass, CharacterClass::identifierStart))

{

state = processLexem(storedLexem, result, StateFlag::none);

**if**(state != StateFlag::error && state != StateFlag::criticalError)

{

state = StateFlag::identifier;

startingCol = col;

startingRow = row;

storedLexem = "";

storedLexem += currentChar;

}

}

**break**;

**default**:

**break**;

}

}

}

**if**(state == StateFlag::comment)

{

errorMessage = "Error: unclosed comment. Missing \*) at the end of file\n";

state = StateFlag::criticalError;

}

**else** **if**(state != StateFlag::none && state != StateFlag::criticalError)

{

state = processLexem(storedLexem, result, state);

}

**if** (state == StateFlag::criticalError || state == StateFlag::error)

{

std::cerr << errorMessage << std::endl;

}

**return** result;

}

Lexer::StateFlag Lexer::processLexem(std::string& lexem, LexemString& buffer, StateFlag expected)

{

StateFlag result = StateFlag::error;

**auto** search = tables[Configuration::TableID::keywords]->find(lexem);

**switch**(expected)

{

**case** StateFlag::identifier:

**if**(search.found == SearchResult::found)

{

buffer.push\_back(EncodedLexem(lexem, search.code, startingRow, startingCol));

result = StateFlag::none;

}

**else** **if**((search = tables[Configuration::TableID::identifiers]->find(lexem)).found == SearchResult::found)

{

buffer.push\_back(EncodedLexem(lexem, search.code, startingRow, startingCol));

result = StateFlag::none;

}

**else**

{

**int** code = tables[Configuration::TableID::identifiers]->getSize() + config.identifierDiapasonStart;

tables[Configuration::TableID::identifiers]->add(lexem, code);

buffer.push\_back(EncodedLexem(lexem, code, startingRow, startingCol));

result = StateFlag::none;

}

**break**;

**case** StateFlag::constant:

search = tables[Configuration::TableID::constants]->find(lexem);

**if**(search.found == SearchResult::found)

{

buffer.push\_back(EncodedLexem(lexem, search.code, startingRow, startingCol));

result = StateFlag::none;

}

**else**

{

**int** code = tables[Configuration::TableID::constants]->getSize() + config.constantDiapasonStart;

tables[Configuration::TableID::constants]->add(lexem, code);

buffer.push\_back(EncodedLexem(lexem, code, startingRow, startingCol));

result = StateFlag::none;

}

**break**;

**case** StateFlag::delimiter:

search = tables[Configuration::TableID::delimiters]->find(lexem);

**if**(search.found == SearchResult::found)

{

**if**(search.code == config.commentStart)

{

result = StateFlag::comment;

}

**else** **if**(search.code == config.commentEnd)

{

result = StateFlag::criticalError;

**char** cMessage[250];

sprintf(cMessage, "Error. Encountered \*) without preceding (\* at %i; %i", startingRow, startingCol);

errorMessage = std::string(cMessage);

}

**else**

{

buffer.push\_back(EncodedLexem(lexem, search.code, startingRow, startingCol));

result = StateFlag::none;

}

lexem = "";

}

**else** **if**(search.found == SearchResult::notFound)

{

**if**(preservedLexem != "")

{

StateFlag checkState = processLexem(preservedLexem, buffer, StateFlag::none);

preservedLexem = "";

lexem = lexem.substr(lexem.length()-1);

startingCol += lexem.length()-1;

**if**(checkState == config.commentStart)

{

result = StateFlag::comment;

lexem = "";

}

**else** **if**(search.code == config.commentEnd)

{

result = StateFlag::criticalError;

**char** cMessage[250];

sprintf(cMessage, "Error. Encountered \*) without preceding (\* at %i; %i", startingRow, startingCol);

errorMessage = std::string(cMessage);

}

**else**

{

search = tables[Configuration::TableID::delimiters]->find(lexem);

**if**(search.found == SearchResult::found)

{

buffer.push\_back(EncodedLexem(lexem, search.code, startingRow, startingCol));

result = StateFlag::none;

lexem = "";

}

**else** **if**(search.found == SearchResult::notFound)

{

lexem = "";

result = StateFlag::criticalError;

**char** cMessage[250];

sprintf(cMessage, "Unrecognized delimiter %s at %i; %i", lexem.c\_str(), startingRow, startingCol);

errorMessage = std::string(cMessage);

}

**else** **if**(search.found == SearchResult::midWayNotFound)

{

result = StateFlag::delimiter;

}

**else** **if**(search.found == SearchResult::midWayFound)

{

preservedLexem = lexem;

result = StateFlag::delimiter;

}

}

}

**else**

{

result = StateFlag::criticalError;

**char** cMessage[250];

sprintf(cMessage, "Unrecognized delimiter %s at %i; %i", lexem.c\_str(), startingRow, startingCol);

errorMessage = std::string(cMessage);

}

}

**else** **if**(search.found == SearchResult::midWayNotFound)

{

result = StateFlag::delimiter;

}

**else** **if**(search.found == SearchResult::midWayFound)

{

**if**(search.code == config.commentStart)

{

result = StateFlag::comment;

lexem = "";

}

**else**

{

preservedLexem = lexem;

result = StateFlag::delimiter;

}

}

**break**;

**case** StateFlag::none:

search = tables[Configuration::TableID::delimiters]->find(lexem);

**if**(search.found == SearchResult::found || search.found == SearchResult::midWayFound)

{

**if**(search.code == config.commentStart)

{

result = StateFlag::comment;

lexem = "";

}

**else**

{

buffer.push\_back(EncodedLexem(lexem, search.code, startingRow, startingCol));

result = StateFlag::none;

}

}

**else**

{

result = StateFlag::criticalError;

**char** cMessage[250];

sprintf(cMessage, "Unrecognized delimiter %s at %i; %i", lexem.c\_str(), startingRow, startingCol);

errorMessage = std::string(cMessage);

}

**break**;

**default**:

result = StateFlag::error;

errorMessage = "Unrecognized token ";

errorMessage += lexem;

**break**;

}

**return** result;

}

**Main.cpp**

#include <unordered\_map>

#include <vector>

#include <iostream>

#include <fstream>

#include <string.h>

# include "Lexer.h"

Configuration readConfiguration(std::ifstream& reader)

{

Configuration result;

std::string s;

for(int i = 0; i < Configuration::TableID::TABLE\_COUNT; i++)

{

std::getline(reader, result.tables[i]);

}

std::getline(reader, s);

result.constantDiapasonStart = atoi(s.c\_str());

std::getline(reader, s);

result.identifierDiapasonStart = atoi(s.c\_str());

std::getline(reader, s);

result.commentStart = atoi(s.c\_str());

std::getline(reader, s);

result.commentEnd = atoi(s.c\_str());

return result;

}

std::ifstream getInputStream(int argc, char\*\* argv)

{

std::ifstream result(argv[1]);

return result;

}

std::ostream& getOutputStream(int argc, char\*\* argv)

{

std::ostream\* resultPointer;

if(argc > 2)

{

if(!strcmp(argv[2], "std"))

{

resultPointer = &std::cout;

}

else

{

resultPointer = new std::ofstream(argv[2]);

}

}

else

{

resultPointer = &std::cout;

}

return \*resultPointer;

}

std::ifstream getConfStream(int argc, char\*\* argv)

{

std::ifstream reader;

if(argc <= 3)

{

reader = std::ifstream("tables\\defaultConfig.conf");

}

else if(!strcmp(argv[3], "default"))

{

reader = std::ifstream("tables\\defaultConfig.conf");

}

else

{

reader = std::ifstream(argv[3]);

}

return reader;

}

void outputResults(Lexer& l, LexemString& str, std::ostream& stream)

{

for(unsigned int i = 0; i < str.size(); i++)

{

stream << str[i].code << " - " << str[i].lexem << std::endl;

}

stream << "--------Identifiers--------\n";

l.getTable(Configuration::TableID::identifiers)->put(stream);

stream << "--------Keywords--------\n";

l.getTable(Configuration::TableID::keywords)->put(stream);

stream << "--------Constants--------\n";

l.getTable(Configuration::TableID::constants)->put(stream);

stream << std::endl;

}

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

{

std::ifstream configuration = getConfStream(argc, argv);

Configuration conf = readConfiguration(configuration);

configuration.close();

std::ifstream input = getInputStream(argc, argv);

Lexer lexer(conf);

auto lexemString = lexer.processSource(input);

input.close();

std::ostream& output = getOutputStream(argc, argv);

outputResults(lexer, lexemString, output);

if(&output != &std::cout)

{

static\_cast<std::ofstream&>(output).close();

}

return 0;

}

**Тести**

True test 1:

Вихідний код:

123[aBc2]13<=13(\*\*\*) 123 t32;a>=13

a(\*\*)b =(\*

\* aBx\* \* \* \*) <=(\*\*) <=

(\*\*)

Результат:

10001 - 123

42 - [

601 - aBc2

43 - ]

10002 - 13

303 - <=

10002 - 13

10001 - 123

602 - t32

44 - ;

603 - a

304 - >=

10002 - 13

603 - a

604 - b

47 - =

303 - <=

303 - <=

Вихідний код:

PROGRAM yasz

BEGIN

a := 2;

t := a;

a := 4;x>=8;

; ;

.. (\*

123 \*\* + + += -- uar >= <= ; < \*

123

PROGRAM yasz

1 + 1

= 3

; \*\*

\* )

\*)

END

Результат:

401 - PROGRAM

601 - yasz

402 - BEGIN

602 - a

305 - :=

10001 - 2

44 - ;

603 - t

305 - :=

602 - a

44 - ;

602 - a

305 - :=

10002 - 4

44 - ;

604 - x

304 - >=

10003 - 8

44 - ;

44 - ;

44 - ;

49 - .

49 - .

403 – END

False test 1:

Вихідний код:

12t

Результат:

Error: invalid constant format at 1; 3

False test 2:

Вихідний код:

PROGRAM yaz

BEGIN

(\*\*\*

END \* )

Результат:

Error: unclosed comment. Missing \*) at the end of file

401 - PROGRAM

601 - yaz

402 – BEGIN