Alex Ho
CS 323 - Section 1
Prof. Choi
Source Code

```
/*Main file*/
#include<iostream>
#include<string>
#include<fstream>
#include"Lexical.h"
#include"Token.h"
#include"SyntaxAnalyzer.h"
using namespace std;

void optionOne();
void optionTwo();
void testCase(string fileInput);

int main()
{
	cout << "*****************************************\n"
		<< "WEICOME TO CS 323 COMPILER APPLICATION\n"
		<< "*****************************************\n";
	int input = 0;
	do {
			cout<< "Menu\n"
			<< "1. Write your own test case.\n"
			<< "2. Run 3 test cases\n"
			<< "3. Exit Application\n\n"
			<< "Please enter number option: ";
		cin >> input;
		cin.clear();
		cin.ignore(1000, '\n');

		while (input > 3 || input < 1)
		{
			cout << "\nPlease reenter number option: ";
			cin >> input;
			cin.clear();
			cin.ignore(1000, '\n');
		}

		switch (input)
		{
			case 1:
				optionOne();
				break;
			case 2:
				optionTwo();
				break;
			case 3:
				cout << "Thank you for using program, have a great day! :)" << endl;
				break;
			default:
				break;
		}
		cout << endl;
	} while (input != 3);

	cout << endl;
	system("pause");
	return 0;
}

void optionOne()
```

```
{

	Lexical lexi;
	string inputLine;
	cout << "Please write underneath one line test case for compiler." << endl;
	getline(cin, inputLine);
	lexi.lexer(inputLine);
	cout << "Here your output for token and lexeme:\n";
	SyntaxAnalyzer syntax(lexi.getListToken());
	syntax.printListSyntax();
	cout << "\nNOTE: The compiler also write out to Output.txt,\n"
		<< "check it out at /Project Compiler CS 323/ Project Compiler/ Output.txt" << endl;
	cout << endl;
}

void optionTwo()
{

	int level = 0;
	cout << "Which test case would you like to see?\n"
		<< "1. Level 1\n"
		<< "2. Level 2\n"
		<< "3. Level 3\n"
		<< "Please enter number option: ";
	cin >> level;
	cin.clear();
	cin.ignore(1000, '\n');

	while (level > 3 || level < 1)
	{
		cout<< "\nPlease reenter number option: ";
		cin >> level;
		cin.clear();
		cin.ignore(1000, '\n');
	}

	switch (level)
	{
	case 1:
		testCase("test1.txt");
		break;

	case 2:
		testCase("test2.txt");
		break;

	case 3:
		testCase("test3.txt");
		break;

	default:
		break;
	}
	if(level == 1 || level == 2 || level == 3)
	cout << "\nNOTE: The compiler also write out to Output.txt,\n"
		<< "check it out at /Project Compiler CS 323/ Project Compiler/ Output.txt" << endl;
}

void testCase(string fileInput)
{
	string currentLine;
	Lexical l;
	fstream inFile;

	inFile.open(fileInput, ios::in);
	cout << endl;
```

```
	while (getline(inFile, currentLine))
	{
		cout << currentLine << endl;
		l.lexer(currentLine);
	}
	cout << endl;
	SyntaxAnalyzer syntax(l.getListToken());
	syntax.printListSyntax();
	inFile.close();
}

/*Lexical.h*/

#pragma once
#ifndef LEXICAL_H
#define LEXICAL_H

#include <iostream>
#include <string>
#include <iomanip>
#include <fstream>
#include <queue>
#include "Token.h"
#include "SyntaxAnalyzer.h"
using namespace std;



//accept 1,2
//input letter, digit, and unknown
//Regular Expression = l|l(l|d)*l
// Note: "->" mean go to next state
const int idFsmTable [5][3] = { {1,4,4}, //state 0, starting state, accepts only letter as input
                                {2,3,4 }, //state 1, letter->2,digit->3, unknown->4
                                {2,3,4},       //state 2, letter->2,digit->3, unknown->4
                                {2,3,4},       //state 3, letter->2,digit->3, unknown->4
                                {4,4,4} };   //state 4, letter->4,digit->4, unknown->4

enum mColIdFsmTable {Letter,Digit,Unknown}; //Column id for FSM Identifier

//accept 1,3
// Regular Expression = d.d
//input digit, '.', and unknown
// Note: "->" mean go to next state
const int intRealFsmTable[5][3] = { {1,4,4}, //state 0, starting state, accepts only digit as input
                                    {1,2,4}, //state 1, ditgit->1,decimal->2,
unknown->4
                                    {3,4,4}, //state 2, ditgit->3,decimal->4,
unknown->4
                                    {3,4,4}, //state 3, ditgit->3,decimal->4,
unknown->4
                                    {4,4,4} }; //state 4, ditgit->4,decimal->4,
unknown->4

//Column id for FSM Digits
enum mColIntRealFsmTable{DigitReal,Decimal}; //Note Unknown already been created in mColIdFsmTable

//Array of string keywords
const string keywords[14] = { "int","real","boolean","if","else","ifend","while",
                              "return","get","put","whileend","function","true","false" };
//Array of string operator
const string operators[11] = { "=","+","-","*","/","=>","=<","^=","==",">","<" };
//Arrat of strig seperator
const string seperators[9] = { "$$", ",", ":", ";", "|", "(", ")","{", "}"};

class Lexical {
```

```
public:
		Lexical() { mStillInComment = false; mLineNumber = 0; }//Constructor
		//Lexical operator= (Lexical const & object);

		bool isOperator(string token);//Check whether string is operator
		bool isSeparator(string token); //Check whether string is separator
		bool isKeyword(string token);		//check whether string is keyword
		bool isReal(string token);				//check whether string is real
		bool isLetterFirst(string token);	//check whether first character of string is letter

		int nextStateIdFsm(char input) const;//check input to determine next state of Identifier FSM
		int nextStateIntRealFsm(char input) const; //check input to determine next state of digit FSM

		/*Lexer function to scanner one string line
		  Determine tokens by charactor
		  Put into a new member token
		  Push that member token into queue list*/
		void lexer(string);

		//Scanner to decide whether comment have done
		void commentScanner(string);

		//Print list of token and lexeme in console and out put into a new file name Output

		queue<Token> getListToken();
		~Lexical();
private:

		//queue<Token> mlistToken; //queue list Token
		bool mStillInComment;
		int mLineNumber;
		Token mFirstTokenInList;
		queue <Token> mListToken;
};



#endif #pragma once

/*Lexical.cpp*/
#include "Lexical.h"
#include "Token.h"

/*Decleration of token functions and constructor*/


//Check whether string is operator
bool Lexical::isOperator(string token)
{
		for (int i = 0; i < 11; i++)
		{
				if (token == operators[i])
				{
						return true;
				}
		}
		return false;
}

//Check whether string is separator
bool Lexical::isSeparator(string token)
{
		for (int i = 0; i < 9; i++)
		{
```

```
                if (token == seperators[i])
                {
                        return true;
                }
        }
        return false;
}

//check whether string is keyword
bool Lexical::isKeyword(string token)
{
        for (int i = 0; i < 14; i++)
        {
                if (token == keywords[i])
                {
                        return true;
                }
        }
        return false;
}

//check whether string is real
bool Lexical::isReal(string token)
{
        for (int i = 0; i < token.length(); i++)
        {                       //if decimal found in string, return true
                if (token[i] == '.')
                {
                        return true;
                }
        }
        return false;
}

//check whether first character of string is letter
bool Lexical::isLetterFirst(string token)
{
        if (isalpha(token[0]))
        {
                return true;
        }

        return false;
}

//check input to determine next state of Identifier FSM
int Lexical::nextStateIdFsm(char input) const
{


        if (isalpha(input))
        {
                return Letter;
        }
        else if (isdigit(input))
        {
                return Digit;
        }
        else
        {
                return Unknown;
        }

}

//check input to determine next state of digit FSM
```

```
int Lexical::nextStateIntRealFsm(char input) const
{


	if (isdigit(input))
	{
		return DigitReal;
	}
	else if (input == '.')
	{
		return Decimal;
	}
	else
	{
		return Unknown;
	}


}

/*Lexer function to scanner one string line
	Determine tokens by charactor
	Put into a new member token
	Push that member token into queue list*/
void Lexical::lexer(string lineInput)
{

	if (mStillInComment) //check if it is still in comment
	{
		commentScanner(lineInput);//using scanner to scan the comment line
	}
	else //else it not comment
	{
		bool terminal = false; //terminal for case stop the loop
		int	currentState = 0; //current state of FSM
		int	currentColumn = 0; // current column of FSM
		int indexChar = 0;
		char currentChar = lineInput[indexChar]; //current single charractor

		//Parses character by character
		while (currentChar != '\0' && !mStillInComment)
		{

			if (currentChar == '[') { //checks whether it is comment character '['
				//scan through the comment line
				while (currentChar != ']' && currentChar != '\0')
				{
					currentChar = lineInput[indexChar];
					indexChar++;
				}

				if (currentChar != ']') //if the comment has not finished then set mStillInComment is true, to keep check
comment next line
				{
					mStillInComment = true;
				}
				else
				{
					currentChar = lineInput[indexChar];
				}
			}
			else //else get the token for lexical analyzer
			{
				if (!isspace(currentChar)) //skip the spaces
				{
					currentState = 0;
```

```
string currentToken = ""; //reset current token
currentToken += currentChar;

if (isalpha(currentChar) || isdigit(currentChar))//If it is letter or digit
{
	terminal = false;//reset terminal
	//While it is not space and terminal is true and character is letter or digit or decimal or '$'
	while (!isspace(currentChar) && !terminal && (isalpha(currentChar) || isdigit(currentChar) || currentChar == '.' || currentChar == '$'))
	{
		//if next charactor is not end line and it is letter or digit or decimal
		if (lineInput[indexChar + 1] != '\0' && (isalpha(lineInput[indexChar + 1]) || isdigit(lineInput[indexChar + 1]) || lineInput[indexChar + 1] == '.'))
		{

			currentChar = lineInput[++indexChar];
			currentToken += currentChar;
		}
		else
		{
			terminal = true;//set terminal to true to get current token and stop the loop
		}
	}

	if (isLetterFirst(currentToken))// if it is letter
	{
		if (isKeyword(currentToken))
		{
			mListToken.push(Token("Keyword", currentToken, mLineNumber, currentState)); //push into queue list
		}
		else
		{
			for (int i = 0; i < currentToken.length();i++)//use FSM to decide if it is accepted value
			{
				currentState = idFsmTable[currentState][nextStateIdFsm(currentToken[i])];
			}

			if (currentState == 1 || currentState == 2)
			{
				mListToken.push(Token("Identifier", currentToken,  mLineNumber, currentState));//push into queue list
			}
			else
			{
				mListToken.push(Token("Unknown", currentToken,  mLineNumber, currentState));//push into queue list

			}
		}

	}
	else //else it is digit
	{

		for (int i = 0; i < currentToken.length();i++)//use FSM to decide if it is accepted value
		{
			currentState = intRealFsmTable[currentState][nextStateIntRealFsm(currentToken[i])];
		}
```

```
                                                                        if (currentState == 1)
                                                                        {
                                                                                mListToken.push(Token("Integer", currentToken,
mLineNumber, currentState));//push into queue list

                                                                        }
                                                                        else if (currentState == 3)
                                                                        {
                                                                                mListToken.push(Token("Real", currentToken,
mLineNumber, currentState));//push into queue list

                                                                        }
                                                                        else
                                                                        {
                                                                                mListToken.push(Token("Unknown", currentToken,
mLineNumber, currentState));//push into queue list

                                                                        }
                                                                }



                                                        }
                                                        else //else it must be separator or operator or unknown
                                                        {

//*************************************************************************************************************
                                                                if (currentToken == "$" && lineInput[indexChar + 1] == '$')//It might be "$$", so
add one more $ to get its token
                                                                {
                                                                        currentChar = lineInput[++indexChar]; //move to next character
                                                                        currentToken += currentChar;//add to current token
                                                                }
                                                                else if(currentToken == "=" &&
                                                                        (lineInput[indexChar + 1] == '='|| lineInput[indexChar + 1] ==
'>'||lineInput[indexChar + 1] == '<')) //It might be comparision, so add one more character to get its token
                                                                {
                                                                        currentChar = lineInput[++indexChar]; //move to next character
                                                                        currentToken += currentChar;//add to current token
                                                                }
                                                                else if (currentToken == "^" && lineInput[indexChar + 1] == '=')//It might be not
equal sign, so add one more character to get its token
                                                                {
                                                                        currentChar = lineInput[++indexChar]; //move to next character
                                                                        currentToken += currentChar;//add to current token
                                                                }

//*************************************************************************************************************

                                                                if (isSeparator(currentToken))//if it is Separator
                                                                {
                                                                        mListToken.push(Token("Separator", currentToken,  mLineNumber,
currentState));//push into queue list

                                                                }
                                                                else if (isOperator(currentToken))//if it is Operator
                                                                {
                                                                        mListToken.push(Token("Operator", currentToken,  mLineNumber,
currentState));//push into queue list
                                                                }
                                                                else// else it is unkown
                                                                {
                                                                        mListToken.push(Token("Unknown", currentToken,  mLineNumber,
currentState));//push into queue list
                                                                }
```

```
                                                }

                                        }
                                        currentChar = lineInput[++indexChar]; //go to next charactor in string

                                }

                        }

                }

                mLineNumber++; //increase line number counter
}

//Scanner to decide whether comment have done
void Lexical::commentScanner(string lineInput)
{
                        int indexChar = 0;
                        char currentChar = lineInput[indexChar];

                                while (currentChar != ']' && currentChar != '\0') // loop continues until the end of line or comment ']'
                                {
                                        currentChar = lineInput[indexChar++];
                                }

                                if (currentChar == ']')//if it is end of comment set mStillInComment to false
                                        mStillInComment = false;

}


queue<Token> Lexical::getListToken()
{
                return mListToken;
}

Lexical::~Lexical()
{
                if (!mListToken.empty())
                {
                        while (!mListToken.empty())
                        {
                                mListToken.pop();
                        }
                }
}



/*Token.h*/
#pragma once
#ifndef TOKEN_H
#define TOKEN_H
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;
/* Token class is store the token values for lexical analyzer */

class Token
{
                friend ostream & operator<<(ostream &, const Token);

public:
                Token() { Token("na", "na","", 0, 0); } //default constructor
```

```
		Token(string toke, string lexeme, int lineNum, int id); // overload constructor
		Token(string toke, string lexeme, string rule, int lineNum, int id);
		//Overload operator
		Token & operator = (Token const & object);

		//Retriever Token values
		void setToken(string);
		void setLexeme(string);
		void setLineNum(int);
		void setFsmId(int);
		void setProductionRule(string);

		//accessor Token values
		int getFsmId()const;
		int getLineNum()const;
		string getToken()const;
		string getLexeme()const;
		string getProductionRule()const;

		~Token(); //Destructor

private:
		string mToken; //token name
		string mLexeme; // token lexeme
		int mLineNum; //line
		int mNumState;
		string mProductionRule;

};


#endif
/*Token.cpp*/

#include"Token.h"
#include<iostream>
#include<string>

/*Decleration of token functions and constructor*/

Token::Token(string toke, string lexeme, int lineNum, int id)
{
		setToken(toke);
		setLexeme(lexeme);
		setLineNum(lineNum);
		setFsmId(id);
}

Token::Token(string toke, string lexeme, string rule, int lineNum, int id)
{
		Token(toke, lexeme, lineNum, id);
		setProductionRule(rule);
}

Token & Token::operator=(Token const & object)
{
		mToken = object.mToken;
		mLexeme = object.mLexeme;
		mNumState = object.mNumState;
		mLineNum = object.mLineNum;
		mProductionRule = object.mProductionRule;

		return *this;
}

void Token::setToken(string toke) { mToken = toke; }
```

```
void Token::setLexeme(string lexe) { mLexeme = lexe; }
void Token::setLineNum(int num) { mLineNum = num; }
void Token::setFsmId(int id) { mNumState = id; }

void Token::setProductionRule(string rule)
{
	mProductionRule = rule;
}

int Token::getFsmId()const { return mNumState; }
int Token::getLineNum()const { return mLineNum; }
string Token::getToken()const { return mToken; }
string Token::getLexeme()const { return mLexeme; }

string Token::getProductionRule() const
{
	return mProductionRule;
}

Token::~Token(){}

ostream & operator<<(ostream & output, const Token toke)
{
	output  <<"Token: " << left << setw(15) << toke.getToken()  << "Lexeme: "<<toke.getLexeme() << endl
		<< toke.getProductionRule()<<endl;

	return output;
}
```

/*SyntaxAnalyzer.h*/

```
#pragma once
#ifndef SYNTAX_ANALYZER_H
#define SYSTAX_ANALYZER_H

#include"Lexical.h"
#include <queue>
#include <string>
#include <ostream>
#include<fstream>

using namespace std;
/*List qualifiers and relops*/
const string qualifierList[5] = { "real","boolean","int" };
const string relopList[6] = { "==","^=",">","<","=>","=<" };



class SyntaxAnalyzer
{


public:
	/*Default contructor for SyntaxAnalyzer */
	SyntaxAnalyzer();
	/*Overload constructor for SystaxAnalyzer
	parameter: list token*/
	SyntaxAnalyzer(queue <Token> list);
	/*Destructor for SyntaxAnalyzer*/
	~SyntaxAnalyzer();
	/*Error message printer
	parameter: string of error, string of rule name*/
	void errorMessage(string,string);
	/**/
	void nextToken();
	//friend ostream & operator<<(ostream & out, SyntaxAnalyzer &);
```

```
		void printListSyntax();

		/*Syntax rule functions*/
		void Rat18F();
		void OptFunctionDefinition();
		void FunctionDefinition();
		void Function();
		void OptParameterList();
		void ParameterList();
		void Parameter();
		void Qualifier();
		void Body();
		void OptDeclarationList();
		void DeclarationList();
		void Declaration();
		void IDs();
		void StatementList();
		void Statement();
		void Compound();
		void Assign();
		void If();
		void Return();
		void Print();
		void Scan();
		void While();
		void Condition();
		void Relop();
		void Expression();
		void ExpressionPrime();
		void Term();
		void TermPrime();
		void Factor();
		void Primary();
		void Empty();



private:

		string mProductionRules;//contain current production rules
		Token mCurrentTokenObject;// contain current token object of list
		string mCurrentLexeme;//contain current lexeme
		string mCurrentToken;//contain current token
		queue <Token> mListToken;//this list is used to get from the list of lexical analyzer
		queue <Token> mListTokenSyntax;// final list after syntax analyzer

};



#endif // !SYNTAX_ANALYZER_H

/*SyntaxAnalyzer.cpp*/
#include "SyntaxAnalyzer.h"


/*Default constructor; initial mProductionRule, mCurrentToken, and mCurrentLexeme to empty string.*/
SyntaxAnalyzer::SyntaxAnalyzer()
{
		mProductionRules = "";
		mCurrentToken = "";
		mCurrentLexeme = "";

}
/*Overload constructor; receive parameter is a list from lexical analyzer then transfer tokens to mListToken.
- get and pop the first token in list
```

```
*/
SyntaxAnalyzer::SyntaxAnalyzer( queue <Token>list )
{
	while (!list.empty())
	{
		mListToken.push(list.front());
		list.pop();
	}

	mCurrentTokenObject = mListToken.front();
	mCurrentToken = mCurrentTokenObject.getToken();
	mCurrentLexeme = mCurrentTokenObject.getLexeme();
	mProductionRules = "";
	mListToken.pop();

	Rat18F();
}

SyntaxAnalyzer::~SyntaxAnalyzer()
{}


/*errorMessage(string typeSyntax, string rules):
Parameter: string of type sytax error, string of name rule
Print the error line number, token, lexeme, what missing and rule name
Return nothing but back to menu.
*/
void SyntaxAnalyzer::errorMessage(string typeSyntax, string rules)
{
	printListSyntax();
	cerr << "\nError at line: " << mCurrentTokenObject.getLineNum()
		<< "\nToken: " << mCurrentToken
		<< "\nLexeme: " << mCurrentLexeme
		<< "\nMissing: " << typeSyntax
		<< "\nRule: " << rules
		<< endl;

	return;
}
/*nextToken()
Set production rule for current token object and add to final list
Move on to next token in the list and pop old one
Handle error if list empty then print error message
*/
void SyntaxAnalyzer::nextToken()
{
	mCurrentTokenObject.setProductionRule(mProductionRules);
	mListTokenSyntax.push(mCurrentTokenObject);
	mProductionRules = "";
	if (!mListToken.empty())
	{
		mCurrentTokenObject = mListToken.front();
		mCurrentToken = mCurrentTokenObject.getToken();
		mCurrentLexeme = mCurrentTokenObject.getLexeme();
		mListToken.pop();
	}
	else
		cerr << "Empty List" << endl;

}
/*printListSyntax()
Create or open Output file to write out
Print the list syntax token in the console and write out the file
Close the file
*/
void SyntaxAnalyzer::printListSyntax()
```

```
{
	fstream outFile;
	outFile.open("Output.txt", ios::out | ios::app);
	cout << "*************************************************************" << endl;
	while (!mListTokenSyntax.empty())
	{

		cout << mListTokenSyntax.front();
		outFile <<"Token: " << left << setw(15) << mListTokenSyntax.front().getToken() <<"Lexeme: "<<
mListTokenSyntax.front().getLexeme() << endl;
		outFile << mListTokenSyntax.front().getProductionRule() << endl;

		mListTokenSyntax.pop();

	}

	outFile << endl;
	outFile.close();

}

//R1. <Rat18F>  ::=   <Opt Function Definitions>   $$  <Opt Declaration List>  <Statement List>  $$
void SyntaxAnalyzer::Rat18F()
{
	mProductionRules += "<Rat18F> -> <Opt Function Definitions>   $$  <Opt Declaration List>  <Statement List> $$\n";
	if (mCurrentLexeme == "function")
	{
		OptFunctionDefinition();
		if (mCurrentLexeme == "$$")
		{
			nextToken();
			OptDeclarationList();
			StatementList();
			if (mCurrentLexeme != "$$")
			{
				errorMessage("$$", "<Rat18F>");
			}
		}
		else
		{
			errorMessage("$$", "<Rat18F>");
		}
	}
	else if (mCurrentLexeme == "$$")
	{
		nextToken();
		OptDeclarationList();
		StatementList();
		if (mCurrentLexeme != "$$")
		{
			errorMessage("$$","<Rat18F>");
		}
	}
	else
	{
		errorMessage("$$ or function", "<Rat18F>");
	}

}

//R2. <Opt Function Definitions> ::= <Function Definitions>     |  <Empty>
void SyntaxAnalyzer::OptFunctionDefinition()
{
	mProductionRules += "<OptFunctionDefinition> -> ";

	if (mCurrentLexeme == "function")
```

```
	{
		mProductionRules += "<Function Definitions>\n";
		FunctionDefinition();
	}
	else if (mCurrentLexeme == "$$")
	{
		Empty();

	}
	else
	{
		errorMessage("$$", "<OptFunctionDefinition>");
	}


}

//R3. <Function Definitions>  ::= <Function> | <Function> <Function Definitions>
void SyntaxAnalyzer::FunctionDefinition()
{
	mProductionRules += "<Function Definitions> -> <Function>\n ";

		Function();
		if (mCurrentLexeme == "function")
		{

			FunctionDefinition();
		}

}

//R4. <Function> ::= function  <Identifier>   ( <Opt Parameter List> )  <Opt Declaration List>  <Body>
void SyntaxAnalyzer::Function()
{
	mProductionRules += "<Function> -> function  <Identifier>   ( <Opt Parameter List> )  <Opt Declaration List>  <Body>\n";
	nextToken();
	if (mCurrentToken == "Identifier")
	{
		nextToken();
		if (mCurrentLexeme == "(")
		{
			nextToken();
			OptParameterList();

			if (mCurrentLexeme == ")")
			{
				nextToken();
				OptDeclarationList();
				Body();
			}
			else
				errorMessage(")", "<Function>");

		}
		else
			errorMessage("(", "<Function>");

	}
	else
	errorMessage("Identifier", "<Function>");

}

//R5. <Opt Parameter List> ::=  <Parameter List>    |    <Empty>
void SyntaxAnalyzer::OptParameterList()
{
```

```
	mProductionRules += "<OptParameterList> -> ";

	if (mCurrentToken == "Identifier")
	{
		mProductionRules += "<Parameter List> \n";
		ParameterList();
	}
	else if (mCurrentLexeme == ")")
	{
		Empty();
	}
	else
	{
		errorMessage("Identifier or )","<OptParameterList>");
	}

}

//R6. <Parameter List>  ::=  <Parameter>    |     <Parameter> , <Parameter List>
void SyntaxAnalyzer::ParameterList()
{
	mProductionRules += "<ParameterList> -> <Parameter> \n";


	if (mCurrentToken == "Identifier")
	{
		Parameter();

		if (mCurrentLexeme == ",")
		{
			nextToken();
			ParameterList();
		}

	}
	else
	{
		errorMessage("Identifier", "<ParameterList>");
	}
}

//R7. <Parameter> :: = <IDs > : <Qualifier>
void SyntaxAnalyzer::Parameter()
{
	mProductionRules += "<Parameter> ->  <IDs > : <Qualifier> \n";

	if (mCurrentToken == "Identifier")
	{
		IDs();

		if (mCurrentLexeme == ":")
		{
			nextToken();
			Qualifier();
		}
		else
		{
			errorMessage(":", "<Parameter>");
		}
	}
	else
	{
		errorMessage("Identifier", "<Parameter>");
	}
}
```

```
//R8. <Qualifier> ::= int    |    boolean    | real
void SyntaxAnalyzer::Qualifier()
{

	bool isFound = false;
	for (string str : qualifierList)
	{
		if (mCurrentLexeme == str)
		{
			mProductionRules += "Qualifier -> " + str +"\n";
			isFound = true;
		}
	}
	if (!isFound)
	{
		errorMessage("int or boolean or real ","<Qualifier>");
	}

	nextToken();
}

//R9. <Body>  ::=  {  < Statement List>  }
void SyntaxAnalyzer::Body()
{
	mProductionRules += "<Body> -> {  < Statement List>  }\n";

	if (mCurrentLexeme == "{")
	{
		nextToken();
		StatementList();

		if (mCurrentLexeme == "}")
		{
			nextToken();
		}
		else
			errorMessage("}","<Body>");

	}
	else
		errorMessage("{","<Body>");
}

//R10. <Opt Declaration List> ::= <Declaration List>   |    <Empty>
void SyntaxAnalyzer::OptDeclarationList()
{
	mProductionRules += "<OptDeclarationList> -> ";

	if (mCurrentLexeme == "{"|| mCurrentLexeme == "while"|| mCurrentLexeme == "get"|| mCurrentLexeme == "put"|| mCurrentLexeme ==
"return"
		|| mCurrentLexeme == "return"|| mCurrentToken == "Identifier")
	{
		Empty();
	}
	else if (mCurrentLexeme == "int" || mCurrentLexeme == "boolean" || mCurrentLexeme == "real")
	{
		mProductionRules += "<Declaration List> \n";
		DeclarationList();
	}
	else
	{
		errorMessage("statements or int, boolean, real","<Opt Declaration List>");
	}
}

//R11. <Declaration List>  := <Declaration> ;     |      <Declaration> ; <Declaration List>
```

```
void SyntaxAnalyzer::DeclarationList()
{
	mProductionRules += "<DeclarationList> -> <Declaration> ; \n";


	Declaration();

	if (mCurrentLexeme == ";")
	{

		nextToken();
		if (mCurrentLexeme == "int" || mCurrentLexeme == "boolean" || mCurrentLexeme == "real")
		{

			DeclarationList();
		}
	}
	else
	{
		errorMessage(";","<DeclarationList>");
	}
}

//R12. <Declaration> ::=   <Qualifier > <IDs>
void SyntaxAnalyzer::Declaration()
{
	mProductionRules += "<Declaration> -> <Qualifier > <IDs>  \n";
	Qualifier();
	if (mCurrentToken == "Identifier")
	{
		IDs();
	}
	else
	{
		errorMessage("Identifier", "<Declaration>");
	}
}

//R13. <IDs> ::=     <Identifier>    | <Identifier>, <IDs>
void SyntaxAnalyzer::IDs()
{
	mProductionRules += "<IDs> -> <Identifier> \n";

	if (mCurrentToken == "Identifier")
	{
		nextToken();

		if (mCurrentLexeme == ",")
		{

			nextToken();
			if (mCurrentToken == "Identifier")
			{

				IDs();
			}
		}
	}
	else
		errorMessage("Identifier", "<IDs>");
}

//R14. <Statement List> ::=   <Statement>   | <Statement> <Statement List>
void SyntaxAnalyzer::StatementList()
{
	mProductionRules += "<StatementList> -> <Statement> \n";
```

```
                Statement();
                if (mCurrentLexeme == "{" || mCurrentLexeme == "if" || mCurrentLexeme == "return"
                        || mCurrentLexeme == "put" || mCurrentLexeme == "get" || mCurrentLexeme == "while"
                        || mCurrentToken == "Identifier")
                {
                        StatementList();
                }

}

//R15. <Statement> ::=   <Compound>  |  <Assign>  |   <If>  |  <Return>   | <Print>   |   <Scan>   |  <While>
void SyntaxAnalyzer::Statement()
{
        mProductionRules += "<Statement> ->  ";

        if (mCurrentToken=="Identifier")
        {
                mProductionRules += "<Assign>\n";
                Assign();
        }
        else if (mCurrentLexeme == "{")
        {
                mProductionRules += "<Compound>\n";
                Compound();
        }
        else if (mCurrentLexeme == "if")
        {
                mProductionRules += "<If>\n";
                If();
        }
        else if (mCurrentLexeme == "return")
        {
                mProductionRules += "<Return>\n";
                Return();
        }
        else if (mCurrentLexeme == "put")
        {
                mProductionRules += "<Print>\n";
                Print();
        }
        else if (mCurrentLexeme == "get")
        {
                mProductionRules += "<Scan>\n";
                Scan();
        }
        else if (mCurrentLexeme == "while")
        {
                mProductionRules += "<While>\n";
                While();
        }
        else
        {
                errorMessage("{, if, return, put, get, while, identifier","<Statement>");
        }
}

//R16. <Compound> ::=   {  <Statement List>  }
void SyntaxAnalyzer::Compound()
{
        mProductionRules += "<Compound> ->  {  <Statement List>  }\n";

        if (mCurrentLexeme == "{")
        {
                nextToken();
                StatementList();
```

```
                if (mCurrentLexeme == "}")
                {
                        nextToken();
                }
                else
                {
                        errorMessage("}", "<Compound>");
                }
        }
        else
        {
                errorMessage("{", "<Compound>");
        }
}

//R17. <Assign> ::=     <Identifier> = <Expression> ;
void SyntaxAnalyzer::Assign()
{
        mProductionRules += "<Assign> ->   <Identifier> = <Expression> ;\n";

        if (mCurrentToken == "Identifier")
        {
                nextToken();
                if (mCurrentLexeme == "=")
                {
                        nextToken();
                        Expression();
                        if (mCurrentLexeme == ";")
                        {
                                nextToken();
                        }
                        else
                        {
                                errorMessage(";", "<Assign>");
                        }
                }
                else
                {
                        errorMessage("=", "<Assign>");
                }
        }
        else
        {
                errorMessage("Identifier", "<Assign>");
        }

}

//R18. <If> ::=     if  ( <Condition>  ) <Statement>   ifend    |   if (<Condition>) < Statement > else  <Statement>  ifend
void SyntaxAnalyzer::If()
{
        mProductionRules += "<If> -> if  ( <Condition>  ) <Statement>  \n";

        if (mCurrentLexeme == "if")
        {
                nextToken();
                if (mCurrentLexeme == "(")
                {
                        nextToken();
                        Condition();
                        if (mCurrentLexeme == ")")
                        {
                                nextToken();
                                Statement();
                                if (mCurrentLexeme == "endif")
                                {
```

```
                                        mProductionRules += "ifend";
                                        nextToken();
                                }
                                else if (mCurrentLexeme == "else")
                                {
                                        mProductionRules += "else  <Statement>  ifend";
                                        nextToken();
                                        Statement();

                                        if (mCurrentLexeme == "ifend")
                                        {
                                                nextToken();
                                        }
                                        else
                                                errorMessage("ifend","<If>");

                                }
                                else
                                {
                                        errorMessage("endif or else", "<If>");
                                }

                        }
                        else
                        {
                                errorMessage(")", "<If>");
                        }

                }
                else
                {
                        errorMessage("(", "<If>");
                }


        }
        else
        {
                errorMessage("if", "<If>");
        }
}

//R19. <Return> ::=  return ; |  return <Expression> ;
void SyntaxAnalyzer::Return()
{
        if (mCurrentLexeme == "return")
        {
                nextToken();

                if (mCurrentLexeme == "-" || mCurrentLexeme == "int" || mCurrentLexeme == "real" || mCurrentLexeme == "("
                        || mCurrentLexeme == "true" || mCurrentLexeme == "false" || mCurrentToken == "Identifier")
                {
                        mProductionRules += "<Return> -> return <Expression>;\n";
                        Expression();
                        if (mCurrentLexeme == ";")
                        {
                                nextToken();
                        }
                        else
                        {
                                errorMessage(";", "<Return>");
                        }
                }
                else
                {
                        mProductionRules += "<Return> -> return;\n";
```

```
                                        nextToken();

                                }
                        }
                        else
                        {
                                errorMessage("return", "<Return>");
                        }


}

//R20. <Print> ::=    put ( <Expression>);
void SyntaxAnalyzer::Print()
{
                mProductionRules += "<Print> -> put ( <Expression>);\n";
                if (mCurrentLexeme == "put")
                {
                                nextToken();

                                if (mCurrentLexeme == "(")
                                {
                                                nextToken();
                                                Expression();
                                                if (mCurrentLexeme == ")")
                                                {
                                                                nextToken();

                                                                if (mCurrentLexeme == ";")
                                                                {
                                                                                nextToken();
                                                                }
                                                                else
                                                                {
                                                                                errorMessage(";", "<Print>");
                                                                }

                                                }
                                                else
                                                {
                                                                errorMessage(")", "<Print>");
                                                }

                                }
                                else
                                {
                                                errorMessage("(", "<Print>");
                                }
                }
                else
                {
                                errorMessage("put", "<Print>");
                }
}

//R21. <Scan> ::=    get ( <IDs> );
void SyntaxAnalyzer::Scan()
{
                mProductionRules += "<Scan> -> get ( <IDs> );\n";
                if (mCurrentLexeme == "get")
                {
                                nextToken();

                                if (mCurrentLexeme == "(")
                                {
                                                nextToken();
```

```
                        IDs();

                        if (mCurrentLexeme == ")")
                        {
                                nextToken();

                                if (mCurrentLexeme == ";")
                                {
                                        nextToken();
                                }
                                else
                                {
                                        errorMessage(";", "<Scan>");
                                }
                        }
                        else
                        {
                                errorMessage(")", "<Scan>");
                        }
                }
                else
                {
                        errorMessage("(", "<Scan>");
                }
        }
        else
        {
                errorMessage("get", "<Scan>");
        }
}

//R22. <While> ::=  while ( <Condition>  )  <Statement>  whileend
void SyntaxAnalyzer::While()
{
        mProductionRules += "<While> -> while ( <Condition>  )  <Statement>  whileend\n";

        if (mCurrentLexeme == "while")
        {
                nextToken();
                if (mCurrentLexeme == "(")
                {
                        nextToken();
                        Condition();
                        if (mCurrentLexeme == ")")
                        {
                                nextToken();
                                Statement();
                                if (mCurrentLexeme == "whileend")
                                {
                                        nextToken();
                                }
                                else
                                {
                                        errorMessage("whileend", "<While>");
                                }

                        }
                        else
                        {
                                errorMessage(")", "<While>");
                        }

                }
                else
                {
                        errorMessage("(", "<While>");
```

```
                }


        }
        else
        {
                errorMessage("while", "<While>");
        }

}

//R23. <Condition> ::=    <Expression>  <Relop>   <Expression>
void SyntaxAnalyzer::Condition()
{
        mProductionRules += "<Condition> ->  <Expression>  <Relop>   <Expression>\n";
        if (mCurrentLexeme == "-" || mCurrentLexeme == "int" || mCurrentLexeme == "real" || mCurrentLexeme == "("
                || mCurrentLexeme == "true" || mCurrentLexeme == "false" || mCurrentToken == "Identifier")
        {
                Expression();
                Relop();
                Expression();
        }
        else
        {
                errorMessage("first condition", "<Condition>");
        }
}

//R24. <Relop> ::=      ==  |  ^=   |  >   |  <   |  =>   |  =<
void SyntaxAnalyzer::Relop()
{
        bool isFound = false;
        for (string str : relopList)
        {
                if (mCurrentLexeme == str)
                {
                        mProductionRules += "\n <Relop> -> " + str;
                        isFound = true;
                }
        }

        if (!isFound)
        {
                errorMessage(" ==  |  ^=   |  >   |  <   |  =>   |  =< ","<Relop>");
        }

        nextToken();
}

//R25. <Expression>  ::=    <Term><ExpressionPrime>
void SyntaxAnalyzer::Expression()
{
        mProductionRules += "<Expression> -> <Term> <EpressionPrime>\n";
        if (mCurrentLexeme == "-" || mCurrentToken == "Integer" || mCurrentToken == "Real" || mCurrentLexeme == "("
                || mCurrentLexeme == "true" || mCurrentLexeme == "false" || mCurrentToken == "Identifier")
        {
                Term();
                ExpressionPrime();
        }
        else
        {
                errorMessage("first expression ", "<Expression>");
        }
}

//R26. <ExpressionPrime> ::=  + <Term><ExpressionPrime> | - <Term> <ExpressionPrime>  | <Empty>
```

```
void SyntaxAnalyzer::ExpressionPrime()
{
	mProductionRules += "<ExpressionPrime> ->  ";

	if (mCurrentLexeme == "==" || mCurrentLexeme == "^=" || mCurrentLexeme == ">"
		|| mCurrentLexeme == "<" || mCurrentLexeme == "=>" || mCurrentLexeme == "=<" || mCurrentLexeme == ")" ||
mCurrentLexeme == ";")
	{
		Empty();
	}
	else if (mCurrentLexeme == "+" || mCurrentLexeme == "-")
	{

		mProductionRules += mCurrentLexeme + " <Term> <ExpressionPrime>\n";

		nextToken();
		Term();
		ExpressionPrime();
	}
	else
	{

		errorMessage("follow of expression prime", "<ExpressionPrime>");
	}


}

//R27. <Term>    ::=       <Factor> <TermPrime>
void SyntaxAnalyzer::Term()
{
	mProductionRules += "<Term> -> <Factor> <TermPrime>\n";
	if (mCurrentLexeme == "-" || mCurrentToken == "Integer" || mCurrentToken == "Real" || mCurrentLexeme == "("
		|| mCurrentLexeme == "true" || mCurrentLexeme == "false" || mCurrentToken == "Identifier")
	{
		Factor();
		TermPrime();
	}
	else
	{
		errorMessage("first term", "<Term>");
	}
}

//R28. <TermPrime> ::= * <Factor> <TermPrime> | / <Factor> <TermPrime> | <Empty>
void SyntaxAnalyzer::TermPrime()
{
	mProductionRules += "<TermPrime> ->  ";

	if (mCurrentLexeme == "*" || mCurrentLexeme == "/")
	{
		mProductionRules += mCurrentLexeme + "<Factor> <TermPrime> \n";
		nextToken();
		Factor();
		TermPrime();
	}
	else if (mCurrentLexeme == "+" || mCurrentLexeme == "-"|| mCurrentLexeme == "==" || mCurrentLexeme == "^=" || mCurrentLexeme ==
">"
		|| mCurrentLexeme == "<" || mCurrentLexeme == "=>" || mCurrentLexeme == "=<" || mCurrentLexeme == ")" ||
mCurrentLexeme == ";")
	{
		Empty();
	}
	else
	{
		errorMessage("follow term prime", " <TermPrime>");
```

```
		}
}

//R29. <Factor> ::=	-  <Primary>	|	<Primary>
void SyntaxAnalyzer::Factor()
{
		mProductionRules += "<Factor> -> ";

		if (mCurrentLexeme == "-")
		{
				mProductionRules += "-  <Primary>\n";
				nextToken();
				Primary();
		}
		else
		{
				mProductionRules += "<Primary>\n";
				Primary();
		}
}

//R30. <Primary> ::=	<Identifier>  |  <Integer>  |   <Identifier>  ( <IDs> )   |   ( <Expression> )   |  <Real> | true | false
void SyntaxAnalyzer::Primary()
{
		mProductionRules += "<Primary> -> ";

		if (mCurrentToken == "Identifier")
		{
				mProductionRules += "<Identifier> \n";
				nextToken();

				if (mCurrentLexeme == "(")
				{
						mProductionRules += " ( <IDs> )\n";
						nextToken();
						IDs();

						if (mCurrentLexeme == ")")
						{
								nextToken();
						}
						else
								errorMessage(")","<Primary>");
				}

		}
		else if (mCurrentLexeme == "(")
		{
				mProductionRules += " ( <Expression> )\n";
				nextToken();
				Expression();

				if (mCurrentLexeme == ")")
				{
						nextToken();
				}
				else
						errorMessage(")","<Primary>");
		}
		else if (mCurrentToken == "Integer" )
		{
				mProductionRules += " <Integer> \n";

				nextToken();
		}
		else if (mCurrentToken == "Real")
```

```
		{
			mProductionRules += "<Real> \n";
			nextToken();
		}
		else if (mCurrentLexeme == "true" || mCurrentLexeme == "false")
		{
			mProductionRules += mCurrentLexeme+ "\n";
			nextToken();
		}
		else
		{
			errorMessage("first primary", "<Primary>");
		}

}

void SyntaxAnalyzer::Empty()
{
		mProductionRules += "<Empty>\n";
}
```