**CS323 Documentation**

1. **Problem Statement**

he second assignment is to write a syntax analyzer. You may use any top-down parser such as RDP, a predictive recursive descent parser or a table-driven predictive parser.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **How to use your program**

1). Launch Rat20\_Lexer.exe

2). Enter the name of the single file to read.

Files to Read:Example1.txt

Example2.txt

Example3.txt

3). Press yes (Y/y) when prompted, to output the scanned tokens into an output file.

4). Write out the file names to output the entries to

Files to output: output1.txt

output2.txt

output3.txt

5). When prompted to exit or repeat the program, repeat the program by pressing Repeat (N/n). Repeat steps 1 through 5 until finished with all files to read and output.

6). When prompted to exit and you are finished, press Exit (Y/y).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Design of your program**

PART 1: PARSING RULES

R1. <Rat20F> ::= <Opt Function Definitions> $$ <Opt Declaration List> <Statement List> $$

R2. <Opt Function Definitions> ::= <Function Definitions> | <Empty>

R3. <Function Definitions> ::= <Function> | <Function> <Function Definitions>

R4. <Function> ::= function <Identifier> ( <Opt Parameter List> ) <Opt Declaration List> <Body>

R5. <Opt Parameter List> ::= <Parameter List> | <Empty>

R6. <Parameter List> ::= <Parameter> | <Parameter> , <Parameter List>

R7. <Parameter> ::= <IDs > <Qualifier>

R8. <Qualifier> ::= int | boolean | real

R9. <Body> ::= { < Statement List> }

R10. <Opt Declaration List> ::= <Declaration List> | <Empty>

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

R12. <Declaration> ::= <Qualifier > <IDs>

R13. <IDs> ::= <Identifier> | <Identifier>, <IDs>

R14. <Statement List> ::= <Statement> | <Statement> <Statement List>

R15. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>

R16. <Compound> ::= { <Statement List> }

R17. <Assign> ::= <Identifier> = <Expression> ;

R18. <If> ::= if ( <Condition> ) <Statement> fi |

if ( <Condition> ) <Statement> else <Statement> fi

R19. <Return> ::= return ; | return <Expression> ;

R20. <Print> ::= put ( <Expression>);

R21. <Scan> ::= get ( <IDs> );

R22. <While> ::= while ( <Condition> ) <Statement>

R23. <Condition> ::= <Expression> <Relop> <Expression>

R24. <Relop> ::= == | != | > | < | <= | =>

R25. <Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>

R26. <Term> ::= <Term> \* <Factor> | <Term> / <Factor> | <Factor>

R27. <Factor> ::= - <Primary> | <Primary>

R28. <Primary> ::= <Identifier> | <Integer> | <Identifier> ( <IDs> ) | ( <Expression> ) |

<Real> | true | false

R29. <Empty> ::= 

PART 2: PROGRAM DESIGN

Major Components

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Class Token
   1. Has private string variables for lexeme and token type
   2. Has public setter and getter methods, and a print method()
   3. This is handled in Lexer.h

Code: **See Appendix - A (Pg. 9)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Lexer.cpp
   1. Main function contains handling for all execution of program
   2. Utilizes VECTOR<STRING> data structures for throughput of text files, both input and output. Three VECTOR<STRING> structures are used, inputContainer, inputContainerEdit1, and inputContainerEdit2. File input will be placed in inputContainer. These are all passed to Main.cpp to be read and organized
   3. One VECTOR<TOKEN> data structure is used for holding the final throughput of the class TOKEN.
   4. Utilizes For loops three For-Loops for throughput.
      1. First For-Loop edits strings and adds spaces based on addSpaces() function. Throughput is stored in inputContainerEdit1., where it will be tokenized with tokenizer() helper function, and after being tokenized, be stored into inputContainerEdit2..
      2. Second For-Loop utilizes the Lexer() function to take each string in from inputContainer2 and process each string in the vector, and outputs a token then store each output directly into tokenList.
      3. Last For-Loop iterates through tokenList and prints out the lexeme and token type of each token.

Code: **See Appendix - B (Pg. 10)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* 1. bool checkKeywords(string input);
     1. Uses a for loop to iterate through a list of keyword strings, and compare them to a string parameter.
     2. Returns true if found, false if not.

Code: **See Appendix - C (Pg. 11)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* 1. bool checkOperators(string input);
     1. Uses a for loop to iterate through a list of operator strings, and compare them to a string parameter.
     2. Returns true if found, false if not.

Code: **See Appendix - D (Pg. 12)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

g. bool checkSeparators(string input);

* + 1. Uses a for loop to iterate through a list of separator strings, and compare them to a string parameter.
    2. Returns true if found, false if not.

Code: **See Appendix - E (Pg. 12)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. string addSpaces(string input);
   1. Takes a string of input and outputs a string replacement that is the input, but edited with spaces if certain cases are found.
   2. Has a string “” called edit.
   3. Has a char array of characters, cases[], to look for, and a string array containing ++ and --, exceptions[].
   4. Uses a for loop to check for exception cases of ++ and -- and verifies if they are in the current string parameter.
   5. Uses another for loop to iterate through the string, char by char. An if statement first checks if the previous for loop verified the existence of ++ or --. If so, it adds spaces before and after the ++/--, and is added to the edit string.
      1. Else if the char from input is currently not equal to anything in cases[], add the char to the edited string.
      2. Else, a space, the current character, and then another space, are added to the string edit.
   6. After the for loop iterates through the entire string, it returns the edited string.

Code: **See Appendix - F (Pg. 13)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. token lexer(string lexeme);
   1. Takes in a string parameter and returns a token as output.
   2. Implements the DFSM of this assignment.
   3. Firstly checks if the string is an operator or a separator, using helper functions. Returns the token immediately if true, initializing it with the token type of operator or separator, and the lexeme.
   4. It the checks keywords with the regarding helper function and if is true, then immediately returns the token with initialization of the lexeme as the parameter, and the token type of keyword.
   5. The major part of the function is the for loop, which goes through char by char, and based on the DFSM throughput, finally declares the string as a valid token. Then it is processed through two switch statements, one to determine token type, and one to determine if it is valid or not.

Code: **See Appendix - G (Pg. 14)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. vector<string> tokenizer(vector<string> str)
   1. Takes in a vector of strings and returns a vector of strings.
   2. Goes through each string, and parses through each one with strtok\_s(). Using the delimiter “ “ it eventually returns an entire string vector without spaces, and in list format based on vector push\_back().

Code: **See Appendix - H (Pg. 21)**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Parser.cpp

* 1. bool match(string s)
     1. Uses if statements to check end of file markers and keeps track of how many tokens have been parsed by adding to int current
     2. Return if a matching token is found

Code: **See Appendix - I (Pg. 22)**

* 1. bool lookahead(string s)
     1. looks to the next token in the vector in order to create accurate parsing
     2. Also check if we are at the end of the file marker and stopping if so

Code: **See Appendix – J (Pg. 23)**

* 1. Production Rule Void Functions
     1. The rest of the functions are all productions using the above rules given
     2. It uses if statements in order to check the look ahead token and determine what production should be taken according to the rules

Function Example : **See Appendix - K (Pg. 22)**

Main.cpp

* 1. Handles all printing of tokens and lexemes using iterators through the storing Vectors
  2. Also handles all user input regarding printing to .txt files and inputing .txt files using While loops with switch cases determined by user input of Y/N

Code: **See Appendix – L (Pg. 23)**

1. **Any Limitation**

Must have a correctly formatted input function, in accordance to Rat20 rules, to receive correct parsing of tokens

1. **Any shortcomings**

Fixed \*(Reads to end of file fine, then runs into an infinite of printing the end file productions)\*

**Code Appendix:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

class token {

private:

string lexeme = "Null", tokenType = "Null";

public:

// setter methods

void setLexeme(string lex) {

lexeme = lex;

}

void setTokenType(string tok) {

tokenType = tok;

}

// getter methods

string getLexeme() {

return lexeme;

}

string getTokenType() {

return tokenType;

}

// print token Method

void printToken() {

cout << "lexeme: " << lexeme << "\t\tToken Type: " << tokenType << "\n";

}

};

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

int main() {

// Boolean to continue running program unless false

bool alive = true;

// Do-While Loop for program

do {

// Vector String Containers for entire program

vector<string> inputContainer = handleFileInput(); // Processing for file input

vector<string> inputContainerEdit1; // Edit 1 for adding spaces for specific cases to create concise tokenization

vector<string> inputContainerEdit2; // Edit 2 for processing through tokenizer()

// Vector tokenlist

vector<token> tokenList; // Vector of Tokens, to process with lexer and add into vector

char answer = 'x'; // UI interaction inputs with char input

bool printOutBool = false; // Bool to handle errors in user input

// Grab all lines of text and place them into vector string inputContainer

for (auto iterator = inputContainer.begin(); iterator != inputContainer.end(); iterator++) {

//while adding each line, edit each line with addSpaces()

inputContainerEdit1.push\_back(addSpaces(\*iterator));

}

// Separate each string by delimeter " "

inputContainerEdit2 = tokenizer(inputContainerEdit1);

// for each string in Edit2, process through lexer.

// Then output a token, and add to tokenList vector

for (auto iterator = inputContainerEdit2.begin(); iterator != inputContainerEdit2.end(); iterator++) {

tokenList.push\_back(lexer(\*iterator));

}

// print out token information

for (auto iterator = tokenList.begin(); iterator != tokenList.end(); iterator++) {

iterator->printToken();

}

\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

bool checkKeywords(string input) {

vector<string> keywords{

"if", "fi", "else", "while", "for", "return", "get", "double", "int",

"long", "string", "signed", "unsigned", "char", "break", "const", "do", "float",

"sizeof", "void", "class", "struct", "bool", "struct", "enum", "short" };

for (auto iterator = keywords.begin(); iterator != keywords.end(); iterator ++) {

if (\*iterator == input)

return true;

}

return false;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

bool checkOperators(string input)

{

string ops[] = { "++", "--", "==", "<=", ">=", "+=", "-=", "\*=", "<<", ">>", "::", ":", "\*", "+", "-", "=", "/", "<", ">", "=" };

for (int i = 0; i < sizeof(ops) / sizeof(ops[0]); i++) {

if (ops[i] == input)

return true;

}

return false;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

bool checkSeparators(string input)

{

string seps[] = { "(", ")", "$$", "/\*", "\*/", ",", ";", "{", "}" };

for (int i = 0; i < sizeof(seps) / sizeof(seps[0]); i++) {

if (seps[i] == input)

return true;

}

return false;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

string addSpaces(string input) {

// string edit to return the input as an edited string with spaces, if cases and exceptions are found

string edit = "";

size\_t exceptionLocation = 0;

bool exceptionVerified = false;

char cases[] = {'(', ')', ';', ',', '-', '/\*', '\*/', '+', '-' };

string exceptions[] = {"++", "--" };

// Go through special cases, or exceptions, and search if it is present in current string

for (int i = 0; i < exceptions->length(); i++) {

if (input.find(exceptions[i]) != std::string::npos) {

exceptionVerified = true;

exceptionLocation = input.find(exceptions[i]);

}

}

// for each char in string, determine if it is involved in exceptions or cases

// if exceptions or special cases are found, add spaces before and after the case/exception

for (int i = 0; i < input.length(); i++) {

if (i == (int)exceptionLocation && exceptionVerified) { //exceptions block

edit = edit + " " + input[i] + input[i + sizeof(char)] + " ";

i++;

}

else if (input[i] != cases[0] && input[i] != cases[1] //cases block

&& input[i] != cases[2] && input[i] != cases[3]

&& input[i] != cases[4] && input[i] != cases[5]

&& input[i] != cases[6] && input[i] != cases[7]

&& input[i] != cases[8])

{ // if nothing has been found, add char to edit string

edit = edit + input[i];

}

else { // if cases are found but not exceptions, add spaces

edit = edit + " " + input[i] + " ";

}

}

return edit;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

token lexer(string lexeme) {

// Enums to handle final calculation of token type

enum DFSMTokenTypes { INTEGER, REAL, IDENTIFIER, ERROR };

DFSMTokenTypes acceptanceStateType = ERROR;

token currentToken;

// Initialization of start location for analysis and DFSM Starting State

currentToken.setLexeme(lexeme);

char state = 'A';

char currentCharInLexeme = '\0';

bool accepted = true;

// Check if lexeme is an operator or separator

// if either/or, set tokentype then immediately return;

if (checkOperators(lexeme)) {

currentToken.setTokenType("Operator");

return currentToken;

}

if (checkSeparators(lexeme)) {

currentToken.setTokenType("Separator");

return currentToken;

}

// Check if lexeme is a keyword, if so, immediately return token as such

if (checkKeywords(lexeme)) {

currentToken.setTokenType("Keyword");

return currentToken;

}

// Visualization of DFSM

// Primarily unused practically

// Used as visual aid for programming below

char DFSM[8][4]

{ // Accepting states: B, C, D, E, F, H

// Non-accepting states: A, G, Z

// Empty States: Z

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// | L | D | . | \_ |

/\*A\*/{ 'B', 'C', 'Z', 'Z' },

/\*B\*/{ 'D', 'E', 'Z', 'F' },

/\*C\*/{ 'Z', 'C', 'G', 'Z' },

/\*D\*/{ 'D', 'E', 'Z', 'F' },

/\*E\*/{ 'D', 'E', 'Z', 'F' },

/\*F\*/{ 'D', 'E', 'Z', 'F' },

/\*G\*/{ 'Z', 'H', 'Z', 'Z' },

/\*H\*/{ 'Z', 'H', 'Z', 'Z' }

};

// for each char, determine current state

for (int i = 0; i < lexeme.size(); i++) {

currentCharInLexeme = lexeme.at(i);

// If state is A - Starting State // The following states maintain the same structure

if (state == 'A') {

// If L /Letter // Comments for State A will be the same for other states except Z

if (isalpha(currentCharInLexeme)) {

state = 'B';

acceptanceStateType = IDENTIFIER;

accepted = true;

}// If D/number

else if (isdigit(currentCharInLexeme)) {

state = 'C';

acceptanceStateType = INTEGER;

accepted = true;

}// If .

else if (currentCharInLexeme == '.') {

state = 'Z'; //Z

accepted = false;

}// If \_

else if (currentCharInLexeme == '\_') {

state = 'Z'; //Z

accepted = false;

}

else state = 'Z';

}

else if (state == 'B') {

acceptanceStateType = IDENTIFIER; // If DFSM reaches state B, token type will be INTEGER

if (isalpha(currentCharInLexeme)) {

state = 'D';

accepted = true;

}

else if (isdigit(currentCharInLexeme)) {

state = 'E';

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'Z';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'F'; //Z

accepted = true;

}

else state = 'Z';

}

else if (state == 'C') { // If DFSM reaches state C, token type will be INTEGER

acceptanceStateType = INTEGER;

if (isalpha(currentCharInLexeme)) {

state = 'Z'; //Z

accepted = false;

}

else if (isdigit(currentCharInLexeme)) {

state = 'C';

acceptanceStateType = INTEGER;

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'G';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'Z';

accepted = false;

}

else state = 'Z';

}

else if (state == 'D') {

if (isalpha(currentCharInLexeme)) {

state = 'D';

accepted = true;

}

else if (isdigit(currentCharInLexeme)) {

state = 'E';

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'Z';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'F';

accepted = true;

}

else state = 'Z';

}

else if (state == 'E') {

if (isalpha(currentCharInLexeme)) {

state = 'D';

accepted = true;

}

else if (isdigit(currentCharInLexeme)) {

state = 'E';

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'Z';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'F';

accepted = true;

}

else state = 'Z';

}

else if (state == 'F') {

if (isalpha(currentCharInLexeme)) {

state = 'D';

accepted = true;

}

else if (isdigit(currentCharInLexeme)) {

state = 'E';

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'Z';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'F';

accepted = true;

}

else state = 'Z';

}

else if (state == 'G') {

if (isalpha(currentCharInLexeme)) {

state = 'Z';

accepted = false;

}

else if (isdigit(currentCharInLexeme)) {

state = 'H';

acceptanceStateType = REAL;

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'Z';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'Z';

accepted = false;

}

else state = 'Z';

}

else if (state == 'H') {

acceptanceStateType = REAL; // Should DFSM reach state H, token type will be a REAL

if (isalpha(currentCharInLexeme)) {

state = 'Z';

accepted = false;

}

else if (isdigit(currentCharInLexeme)) {

state = 'H';

acceptanceStateType = REAL;

accepted = true;

}

else if (currentCharInLexeme == '.') {

state = 'Z';

accepted = false;

}

else if (currentCharInLexeme == '\_') {

state = 'Z';

accepted = false;

}

else state = 'Z';;

}

else if (state == 'Z') { //Empty states, will not be accepted as valid

accepted = false;

}

}

// Based on Enumerators and whether a certain state was reached

// Initialize token to be returned as the following

switch (acceptanceStateType) {

case INTEGER:

currentToken.setTokenType("Integer");

break;

case REAL:

currentToken.setTokenType("Real");

break;

case IDENTIFIER:

currentToken.setTokenType("Identifier");

break;

default:

currentToken.setTokenType("ERROR");

break;

};

// Based on final state, accept will be true or false

// Acceptance will return current Token as is conceived

// False or default will declare type as invalid

switch (accepted)

{

case true:

return currentToken;

break;

case false:

currentToken.setTokenType("Invalid - False");

return currentToken;

break;

default:

currentToken.setTokenType("Invalid - Default");

return currentToken;

break;

};

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

vector<string> tokenizer(vector<string> str) {

vector<string> output;

for (int i = 0; i < str.size() - 1; i++) {

char\* cstr = (char\*)malloc(1000); // Allocate memory of 1000 for cstr

char\* context = NULL;

strcpy\_s(cstr, 1000, str[i].c\_str());

char\* tok = strtok\_s(cstr, " ", &context); // Tokenize through parsing with delimeter " "

while (tok != 0) {

output.push\_back(tok);

tok = strtok\_s(NULL, " ", &context);

}

free(cstr); // Free memory of cstr

}

return output;

}

bool match(string s) {

if

(current >= tokens.size()) {

error("End of file.");

return false;

}

if (tokens[current].getLexeme() == s) {

current++;

cout << "`------Matched:" << s << endl;

return true;

}

else {

cout << "ERROR: expected: " << tokens[current].getLexeme() << " found: " << s << endl;

current++;

return false;

}

}

void qualifier() {

if (debug) {

cout << "<Qualifier> :: = int | boolean | real" << endl;

}

if (lookahead("Integer"))

match("Integer");

if (lookahead("bool"))

match("bool");

if (lookahead("Real"))

match("Real");

}

K.

cout << "Would you like to print these out to a text file? Y/N\n";

while (printOutBool == false) {

cin >> answer;

switch (answer) {

case 'y':

case'Y':

handleFileOutput(tokenList);

printOutBool = true;

break;

case 'n':

case 'N':

cout << "Have a good day.\n";

printOutBool = true;

break;

default:

cout << "Please enter a valid answer.\n";

break;

}

}