Compiler Final Project Report

CPSC 323-05

Spring 2023

Professor Mohamadreza, Ahmadnia

By: Everette & Abraham

Table of Contents

README…………………………………………………………………….3

The Original Program…………………………………………………4

The Original Grammar………………………………………………..5

The Original Grammar in BNF form……………………………..6

Preparation for the Predictive Parsing Table……………….8

First and Follow Table………………………………………………..11

Predictive Parsing Table……………………………………………..12

Our Compiler Program……………………………………………….16

README

The following report is for the final project for the Compilers and Languages class (CPSC 323-05) at California State University Fullerton taught by Professor Mohmamadreza Ahmadnia in the Spring 2023 semester. Both this report and project were made by Everette and Abraham. The purpose of this project was to make a small compiler to work on a given program, validate or reject it, and convert it to c++.

Below you will find: The Original Program ment to be compiled, The Original Grammar for that program, The Original Grammar converted to BNF for, The Preparation done to the grammar to make a Predictive Parsing Table, The First and Follow table of the prepared grammar, the Predictive Parsing Table, and lastly all the code for the compiler itself.

Inorder to run the code for our compiler a copy has been included in the flash drive associated with this report. The file is called ‘FinalProject.cpp’. For your convenience a ‘run.cmd’ and a ‘run.bash’ file has been included to make compiling and running our code easier. Should both of these fail, simply try compiling and executing our code yourself.

To help test our compiler we have included multiple text files. The file ‘finalp1.txt’ is the original program below. Others are ment to show case the different errors our compiler can catch.

All the documents can be found in the flash drive included with this report and at the GitHub link here: <https://github.com/Digx7/Compilers-And-Languages-Final-Project>

The Original Program

program s2023;

//This program computes and prints the value of a given expression//

var

// declare variables

p1, p2q, pr : integer;

begin

//initialize variables

p1 = 33;

p2q = 412;

pr = p1 + p2q;

display ( pr ); // display pr

// compute the value of the following expression //

pr = p1 \* ( p2q + 2 \* pr) ;

display (, pr ) ; // print the value of pr

end.

The Original Grammar

|  |  |  |
| --- | --- | --- |
| The Original given Grammar | |  |
| <prog> | -> | program <identifier> ; var <dec-list> begin <stat-list> end. |
| <identifier> | -> | <letter>{<letter>|<digit>} |
| <dec-list> | -> | <dec> : <type> ; |
| <dec> | -> | <identifier> , <dec> | <identifier> |
| <type> | -> | integer |
| <stat-list> | -> | <stat> | <stat> <stat-list> |
| <stat> | -> | <write> | <assign> |
| <write> | -> | display ("value", <identifier> ) ; | display ( <identifier> ) ; |
| <assign> | -> | <identifier> = <expr> ; |
| <expr> | -> | <expr> + <term> | <expr> - <term> | <term> |
| <term> | -> | <term> \* <factor> | <term> / <factor> | <factor> |
| <factor> | -> | ( <expr> ) |
| <factor> | -> | <identifier> | <number> |
| <number> | -> | <sign> <digit> {<digit>} |
| <sign> | -> | + | - | lamda |
| <digit> | -> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| <letter> | -> | p | q | r | s |

The Original Grammar in BNF form

|  |  |  |
| --- | --- | --- |
| The Original Grammar in BNF | |  |
| <prog> | -> | program <identifier\_start> ; var <dec-list> begin <stat-list> end. |
| <identifier\_start> | -> | <letter><identifier\_body> |
| <identifier\_body> | -> | <letter><identifier\_body> |
| <identifier\_body> | -> | <digit><identifier\_body> |
| <identifier\_body> | -> | lamda |
| <dec-list> | -> | <dec> : <type> ; |
| <dec> | -> | <identifier\_start> , <dec> |
| <dec> | -> | <idetifier\_start> |
| <type> | -> | integer |
| <stat-list> | -> | <stat> |
| <stat-list> | -> | <stat> <stat-list> |
| <stat> | -> | <write> |
| <stat> | -> | <assign> |
| <write> | -> | display ("value", <identifier\_start> ) ; |
| <write> | -> | display ( <identifier\_start> ) ; |
| <assign> | -> | <identifier\_start> = <expr> ; |
| <expr> | -> | <expr> + <term> |
| <expr> | -> | <expr> - <term> |
| <expr> | -> | <term> |
| <term> | -> | <term> \* <factor> |
| <term> | -> | <term> / <factor> |
| <term> | -> | <factor> |
| <factor> | -> | ( <expr> ) |
| <factor> | -> | <identifier\_start> |
| <factor> | -> | <number\_start> |
| <number\_start> | -> | <sign> <digit> <number\_body> |
| <number\_body> | -> | <digit><number\_body> |
| <number\_body> | -> | lamda |
| <sign> | -> | + |
| <sign> | -> | - |
| <sign> | -> | lamda |
| <digit> | -> | 0 |
| <digit> | -> | 1 |
| <digit> | -> | 2 |
| <digit> | -> | 3 |
| <digit> | -> | 4 |
| <digit> | -> | 5 |
| <digit> | -> | 6 |
| <digit> | -> | 7 |
| <digit> | -> | 8 |
| <digit> | -> | 9 |
| <letter> | -> | p |
| <letter> | -> | q |
| <letter> | -> | r |
| <letter> | -> | s |

Preparation for the Predictive Parsing Table

After getting the Original Grammar in BNF form we had to take the following steps to get it ready for the Predictive Parsing Table.

1. Remove all Left-Hand Recursions
2. Remove all Indeterminates

|  |  |  |
| --- | --- | --- |
| Remove all Left-Hand recursions |  |  |
| <prog> | -> | program <identifier\_start> ; var <dec-list> begin <stat-list> end. |
| <identifier\_start> | -> | <letter> <identifier\_body> |
| <identifier\_body> | -> | <letter> <identifier\_body> |
| <identifier\_body> | -> | <digit> <identifier\_body> |
| <identifier\_body> | -> | lamda |
| <dec-list> | -> | <dec> : <type> ; |
| <dec> | -> | <identifier\_start> , <dec> |
| <dec> | -> | <identifier\_start> |
| <type> | -> | integer |
| <stat-list> | -> | <stat> |
| <stat-list> | -> | <stat> <stat-list> |
| <stat> | -> | <write> |
| <stat> | -> | <assign> |
| <write> | -> | display ( "value" , <identifier\_start> ) ; |
| <write> | -> | display ( <identifier\_start> ) ; |
| <assign> | -> | <identifier\_start> = <expr\_term\_factor\_enter> ; |
| <expr\_term\_factor\_enter> | -> | <term\_enter> <expr> |
| <expr> | -> | + <term\_enter> <expr> |
| <expr> | -> | - <term\_enter> <expr> |
| <expr> | -> | lamda |
| <term\_enter> | -> | <factor> <term> |
| <term> | -> | \* <factor> <term> |
| <term> | -> | /<factor> <term> |
| <term> | -> | lamda |
| <factor> | -> | ( <expr\_term\_factor\_enter> ) |
| <factor> | -> | <identifier\_start> |
| <factor> | -> | <number> |
| <number\_start> | -> | <sign> <digit> <number\_body> |
| <number\_body> | -> | <digit> <number\_body> |
| <number\_body> | -> | lamda |
| <sign> | -> | + |
| <sign> | -> | - |
| <sign> | -> | lamda |
| <digit> | -> | 0 |
| <digit> | -> | 1 |
| <digit> | -> | 2 |
| <digit> | -> | 3 |
| <digit> | -> | 4 |
| <digit> | -> | 5 |
| <digit> | -> | 6 |
| <digit> | -> | 7 |
| <digit> | -> | 8 |
| <digit> | -> | 9 |
| <letter> | -> | p |
| <letter> | -> | q |
| <letter> | -> | r |
| <letter> | -> | s |

|  |  |  |
| --- | --- | --- |
| Remove all Indeterminates |  |  |
| <prog> | -> | program <identifier\_start> ; var <dec-list> begin <stat-list\_enter> end. |
| <identifier\_start> | -> | <letter> <identifier\_body> |
| <identifier\_body> | -> | <letter> <identifier\_body> |
| <identifier\_body> | -> | <digit> <identifier\_body> |
| <identifier\_body> | -> | lamda |
| <dec-list> | -> | <dec\_enter> : <type> ; |
| <dec\_enter> | -> | <identifier\_start> <dec> |
| <dec> | -> | , <dec\_enter> |
| <dec> | -> | lamda |
| <type> | -> | integer |
| <stat-list\_enter> | -> | <stat> <stat-list> |
| <stat-list> | -> | lamda |
| <stat-list> | -> | <stat> <stat-list\_enter> |
| <stat> | -> | <write\_enter> |
| <stat> | -> | <assign> |
| <write\_enter> | -> | display ( <write> |
| <write> | -> | "value" , <identifier\_start> ) ; |
| <write> | -> | <identifier\_start> ) ; |
| <assign> | -> | <identifier\_start> = <expr\_term\_factor\_enter> ; |
| <expr\_term\_factor\_enter> | -> | <term\_enter> <expr> |
| <expr> | -> | + <term\_enter> <expr> |
| <expr> | -> | - <term\_enter> <expr> |
| <expr> | -> | lamda |
| <term\_enter> | -> | <factor> <term> |
| <term> | -> | \* <factor> <term> |
| <term> | -> | /<factor> <term> |
| <term> | -> | lamda |
| <factor> | -> | ( <expr\_term\_factor\_enter> ) |
| <factor> | -> | <identifier\_start> |
| <factor> | -> | <number\_start> |
| <number\_start> | -> | <sign> <digit> <number\_body> |
| <number\_body> | -> | <digit> <number\_body> |
| <number\_body> | -> | lamda |
| <sign> | -> | + |
| <sign> | -> | - |
| <sign> | -> | lamda |
| <digit> | -> | 0 |
| <digit> | -> | 1 |
| <digit> | -> | 2 |
| <digit> | -> | 3 |
| <digit> | -> | 4 |
| <digit> | -> | 5 |
| <digit> | -> | 6 |
| <digit> | -> | 7 |
| <digit> | -> | 8 |
| <digit> | -> | 9 |
| <letter> | -> | p |
| <letter> | -> | q |
| <letter> | -> | r |
| <letter> | -> | s |

First and Follow Table

|  |  |  |
| --- | --- | --- |
| First and Follow |  |  |
| NonTerminal | First | Follow |
| <prog> | Program | end. |
| <identifier\_start> | p q r s | ; ) = , \* / + - : |
| <identifier\_body> | p q r s 0 1 2 3 4 5 6 7 8 9 | ; ) = , \* / + - : |
| <dec-list> | p q r s | begin |
| <dec\_enter> | p q r s | : |
| <dec> | , lamda | : |
| <type> | integer | ; |
| <stat-list\_enter> | display p q r s | end. |
| <stat-list> | lamda display p q r s | end. |
| <stat> | display p q r s | display p q r s end. |
| <write\_enter> | display | display p q r s end. |
| <write> | "value" p q r s | display p q r s end. |
| <assign> | p q r s | display p q r s end. |
| <expr\_term\_factor\_enter> | ( p q r s + - lamda 0 1 2 3 4 5 6 7 8 9 | ;) |
| <expr> | + - lamda | ; ) |
| <term\_enter> | ( p q r s + - lamda 0 1 2 3 4 5 6 7 8 9 | + - ; ) |
| <term> | \* / lamda | + - ; ) |
| <factor> | ( p q r s + - lamda 0 1 2 3 4 5 6 7 8 9 | \* / + - ; ) |
| <number\_start> | + - lamda 0 1 2 3 4 5 6 7 8 9 | \* / + - ; ) |
| <number\_body> | 0 1 2 3 4 5 6 7 8 9 lamda | \* / + - ; ) |
| <sign> | + - lamda | 0 1 2 3 4 5 6 7 8 9 |
| <digit> | 0 1 2 3 4 5 6 7 8 9 | ; ) = , \* / + - : |
| <letter> | p q r s | ; ) = , \* / + - : |

Predictive Parsing Table

Note: due to the massive size of the Predictive Parsing Table it was impossible to get it all on one page. Therefore, it has been broken up into pieces. The entire table reads from left to right, top down of each piece. For example [<prog> , begin] = E\_program is expected. Also, all error messages take the form of E\_x where x is the error message.

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

Our Compiler Program

// Final Project for Compilers

// By: Abraham and Everette

// This program is ment to be a simple compiler

#include <iostream>

#include <fstream>

#include <string>

#include <sstream>

#include <vector>

#include <stack>

#include <algorithm>

#include <stdlib.h>

#include <cstdlib>

using namespace std;

const string LINE1 = "====================================\n";

const string LINE2 = "------------------------------------\n";

const string AUTHORS = "Everette & Abraham";

// UI Functions ===================================

bool getYesOrNoResponse(string exitMessage){

while( 1 > 0){

string input = "";

std::cout << exitMessage << "(y/n): ";

std::cin >> input;

if (input == "N"||input == "n") return false;

else if(input == "Y"||input == "y") return true;

std::cout << "I'm sorry I did not recognize that input\nPlease try again" << std::endl;

}

}

bool doesFileExist(string fileName){

ifstream f(fileName);

return f.good();

}

void welcomeMessage(){

std::cout << LINE1 << "This is a general compiler for the final project using Predictive Parsing Table\n" << "Designed by: " << AUTHORS << "\n" << LINE1 << flush;

}

void goodByeMessage(){

std::cout << "Good Bye :)" << std::endl << flush;

}

string getStringFromUser(string promptMessage){

cout << promptMessage << ": ";

string output;

cin >> output;

return output;

}

string getFileNameFromUser(string promptMessage, string fileExtenstion, bool requireFileToExist = false){

// Initialize variables

string output;

string confirmMessage;

bool confirm;

while(1 > 0){ // will keep looping until the user confirms

output = getStringFromUser(promptMessage);

string fullFileName = output + fileExtenstion;

confirmMessage = "Confirm '" + fullFileName + "' ";

confirm = getYesOrNoResponse(confirmMessage);

if(confirm){

if(requireFileToExist && !doesFileExist(fullFileName)) cout << "I'm sorry " << fullFileName << " was not found" << endl;

else return output;

}

}

}

// Utillity Functions =============================

string addSpaceAroundCharacter(string str, char character, char stoppingCharacter)

{

// Initialize variables

bool canEdit = true;

string output = ""; // a new string we will write to

for(int i = 0; i < str.length(); i++) { // iterates through each character in the string

if(str[i]==stoppingCharacter) // if we see the stoppingCharacter we can't add anymore spaces

canEdit = false;

if(canEdit && str[i]==character) // if we found the character add spaces around it

output = output + " " + str[i] + " ";

else // just copy over the string one for one

output = output + str[i];

}

return output;

}

int findStringInVector(vector<string> mainVector, string stringToLookFor){

// Iterates through our vector looking for stringToLookfor

auto it = find(mainVector.begin(), mainVector.end(), stringToLookFor);

// If element was found

if (it != mainVector.end())

{

// calculating the index of stringToLookFor

int index = it - mainVector.begin();

return index;

}

else {

// If the element is not present in the vector

return -1;

}

}

// Classes =======================================

class language{

public:

vector<vector<string>> table;

vector<string> terminals;

vector<string> nonTerminals;

vector<string> reservedWords;

public:

language(){}

language(vector<vector<string>> \_table, vector<string> \_terminals, vector<string> \_nonTerminals, vector<string> \_reservedWords){

table = \_table;

terminals = \_terminals;

nonTerminals = \_nonTerminals;

reservedWords = \_reservedWords;

}

bool isReservedWord(string input){

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

if(input == reservedWords[i]) return true;

}

return false;

}

string GoTo(string nonTerminal, string terminal){

// Gets the int index of the nonTerminal

int index\_nonTerminal = findStringInVector(nonTerminals, nonTerminal);

// Gets the int index of the terminal

int index\_terminal = findStringInVector(terminals, terminal);

// Checks if either index is out of bounds

if(index\_nonTerminal == -1 || index\_terminal == -1)return "E\_Out of Bounds";

// Returns the value in the table at the given indexes

return table[index\_nonTerminal][index\_terminal];

}

};

class compiler{

private:

stack<string> parsingStack;

string currentInputToken;

string currentPoppedElement;

vector<string> definedIdentifiers;

vector<char> currentIdentifier;

bool isInDebugMode;

bool isLookingAtIdentifier;

bool isDefiningIdentifiers;

int currentLineNumber;

int currentTokenNumber;

int maxNumberOfLines;

public:

language chosenLanguage;

string inputFileName;

vector<string> errors;

private:

void resetCompiler(){

while(!parsingStack.empty())

parsingStack.pop();

currentInputToken = "";

currentPoppedElement = "";

currentLineNumber = 1;

currentTokenNumber = 1;

if(isInDebugMode) cout << "Compiler reset" << endl;

parsingStack.push("<prog>");

if(isInDebugMode) cout << "Push: <prog>" << endl;

}

bool lookingForMatch(string input){

bool matchFound = false;

while(!matchFound){

if(isInDebugMode) cout << LINE2;

//Pop top element from stack

popFromStack();

//if top element == currentInputToken then matchFound = true

if(currentPoppedElement == input){

matchFound = true;

if(isInDebugMode) cout << "MATCH FOUND: " << currentPoppedElement << " == " << input << endl;

if(isLookingAtIdentifier) buildIdentifier(input[0]);

if(currentPoppedElement == "begin") isDefiningIdentifiers = false;

}

else{

string outputString = chosenLanguage.GoTo(currentPoppedElement, input);

if(isInDebugMode) cout << "Go To [" << currentPoppedElement << " , " << input << "] = " << outputString << endl;

if(currentPoppedElement == "<identifier\_start>") isLookingAtIdentifier = true;

if(isLookingAtIdentifier && outputString == "lamda") {

isLookingAtIdentifier = false;

// TODO: Make it so the identifiers are either defined or checked depending on where we are in the code

if(isDefiningIdentifiers) defineIdentifier();

else if (!isDefinedIdentifier()){

outputString = "E\_unknown identifier";

}

}

if(outputString == "") {

outputString = "E\_Unknown error";

}

if(isError(outputString)){

string errorMessage = defineError(outputString, input);

addErrorMessage(errorMessage);

return false;

}

if(outputString != "lamda"){

pushToStack(outputString);

}

}

}

return true;

}

// Parsing Stack functions

bool popFromStack(){

if(parsingStack.empty()) return false;

currentPoppedElement = parsingStack.top();

parsingStack.pop();

if(isInDebugMode) cout << "Pop: " << currentPoppedElement << endl;

return true;

}

void pushToStack(string input){

stack<string> reversingStack;

stringstream s(input);

string token;

while(s >> token) {

reversingStack.push(token);

}

while(!reversingStack.empty()){

parsingStack.push(reversingStack.top());

reversingStack.pop();

if(isInDebugMode) cout << "Push: " << parsingStack.top() << endl;

}

if(isInDebugMode) cout << "Top of Stack is: " << parsingStack.top() << endl;

}

// Identifier functions

bool isDefinedIdentifier(){

string identifier(currentIdentifier.begin(), currentIdentifier.end());

while(!currentIdentifier.empty()) currentIdentifier.pop\_back();

if(findStringInVector(definedIdentifiers, identifier) == -1) return false;

return true;

}

void defineIdentifier(){

string identifier(currentIdentifier.begin(), currentIdentifier.end());

definedIdentifiers.push\_back(identifier);

while(!currentIdentifier.empty()) currentIdentifier.pop\_back();

}

void buildIdentifier(char input){

currentIdentifier.push\_back(input);

if(isInDebugMode) cout << "Building Identifier" << endl;

}

// Error functions

bool isError(string input){

if(input[0] == 'E') return true;

}

string defineError(string inputString, string lastInput){

// if we go out of bounds on our language table

if(inputString == "E\_Out of Bounds"){

// if we tried to look for a terminal in the collumns, which are all non-terminals

if(findStringInVector(chosenLanguage.terminals, currentPoppedElement) != -1){

inputString = "E\_" + currentPoppedElement + " is expected";

}

// if we tried to use a character in our terminals that doesn't exist in our language

else if(findStringInVector(chosenLanguage.terminals, lastInput) == -1){

// when we are at '<type>' the only thing we should see is 'integer'

// so if we see anything else we get this error

if(currentPoppedElement == "<type>")

inputString = "E\_integer is expected";

else

inputString = "E\_" + lastInput + " is an invalid character";

if(lastInput == "e" || lastInput == "E") inputString = inputString + " \ndid you misspell 'end.'";

if(lastInput == "d" || lastInput == "D") inputString = inputString + " \ndid you misspell 'display'";

}

}

// checks if we are at the end of the file

// will trigger if end. is missspelled

if(currentLineNumber == maxNumberOfLines) inputString = "E\_end. is expected";

return inputString;

}

void addErrorMessage(string errorMessage){

string message = errorMessage.substr(2, errorMessage.size() - 2);

string location = "[" + to\_string(currentLineNumber) + "," + to\_string(currentTokenNumber) + "]";

string finalMessage = location + " " + message;

errors.push\_back(finalMessage);

}

public:

compiler(language \_language, string \_inputFileName, int \_maxNumberOfLines){

// Assigning variables

chosenLanguage = \_language;

inputFileName = \_inputFileName;

maxNumberOfLines = \_maxNumberOfLines;

// Ensures these variable start in this state

isInDebugMode = false;

isLookingAtIdentifier = false;

isDefiningIdentifiers = true;

}

bool run(){

resetCompiler();

ifstream MyReadFile(inputFileName);

string myLine;

while(getline (MyReadFile, myLine)){

if(isInDebugMode) cout << LINE1;

if(isInDebugMode) cout << "Looking at line: " << myLine << endl;

if(isInDebugMode) cout << LINE1;

currentTokenNumber = 1;

stringstream s(myLine);

while(s >> currentInputToken){

if(isInDebugMode) cout << "Looking at token: " << currentInputToken << endl;

bool tokenIsReserveWord = chosenLanguage.isReservedWord(currentInputToken);

if(!tokenIsReserveWord){

if(isInDebugMode) cout << "Token: " << currentInputToken << " is NOT a reserved word and is being split up" << endl;

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

string tmp = string(1, currentInputToken[i]);

if(!lookingForMatch(tmp)) return false;

}

}

else{

if(isInDebugMode) cout << "Token: " << currentInputToken << " is a reserved word and is being treated like one character" << endl;

if(!lookingForMatch(currentInputToken)) return false;

}

currentTokenNumber++;

}

currentLineNumber++;

}

MyReadFile.close();

if(currentInputToken == "end."){

return true;

}

else{

// if we got to the end of the file but don't see 'end.'

addErrorMessage("E\_end. is expected");

return false;

}

}

void flushParsingStack(){

if(isInDebugMode) cout << "Items left in parsing stack: " << endl;

while(!parsingStack.empty()){

if(isInDebugMode) cout << parsingStack.top() << endl;

parsingStack.pop();

}

if(isInDebugMode) cout << "End of parsing stack" << endl;

}

void displayAllDefinedIdentifiers(){

cout << "All Defined Identifiers: " << endl;

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

cout << definedIdentifiers[i] << endl;

}

}

void setDebugMode(bool input){

isInDebugMode = input;

}

};

// Part 1 - 4 Functions ==========================

int part1(string readFileName, string writeFileName, bool useDebugMode = false){

// turn the text file "finalp1.txt" into "finalp2.txt"

if(useDebugMode){

cout << LINE1;

cout << "PART 1" << endl;

cout << "Attempting to write a cleaned up version of " << readFileName << " in " << writeFileName << endl;

}

// Open files

ifstream MyReadFile(readFileName);

ofstream MyWriteFile(writeFileName);

// Initialize variables

bool shouldWrite = true;

bool wordWasWritten = false;

int numberOfLines = 0;

string myLine;

while(getline (MyReadFile, myLine)){ // lets us view each line of our read file

// Initialize variables

stringstream s(myLine);

string word;

while(s >> word){ // lets us view each word of each line of our read file

// Initialize variables

string dictonary = ":;,=+-()"; // characters of words to add spaces around

char stoppingChar = '"'; // stopping character so we don't mess up strings

for(int i=0; i<dictonary.length(); ++i){ // for each character in the above dictionary, if its in the word we're looking at then add a space around it

word = addSpaceAroundCharacter(word,dictonary[i],stoppingChar);

}

if(word[0] == '/' && word[1] == '/') shouldWrite = false; // So we don't write comments

if(shouldWrite){ // Writes the current word to the write file

MyWriteFile << word << ' ';

wordWasWritten = true;

}

}

if(wordWasWritten) { // allows us to remove blank lines

MyWriteFile << "\n";

numberOfLines++;

}

// Reset variables for next loop

shouldWrite = true;

wordWasWritten = false;

}

// Closes files

MyReadFile.close();

MyWriteFile.close();

if(useDebugMode){

cout << writeFileName << " should now be a cleaned up version of " << readFileName << endl;

cout << "You can check " << writeFileName << " now or press any key to let the program continue" << endl << flush;

system("PAUSE");

}

return numberOfLines; // Is used the compiler for some error checking

}

vector<string> part2(string orginalFileName ,string readFileName, int maxNumberOfLines, bool useDebugMode = false){

// run the "finalp2.txt" through the grammer to check for errors

if(useDebugMode){

cout << LINE1;

cout << "PART 2" << endl;

cout << "Attempting to trace the cleaned up version of " << orginalFileName << " in " << readFileName << " to see if it is valid" << endl;

cout << "Press any key to beging tracing " << readFileName << endl << flush;

system("PAUSE");

cout << LINE2;

}

// Our langage table

vector<vector<string>> table

{

// program var begin end. integer display ; : , "value=" ( ) = + - \* / 0 1 2 3 4 5 6 7 8 9 p q r s

/\*<prog> \*/{"program <identifier\_start> ; var <dec-list> begin <stat-list\_enter> end.", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected", "E\_program is expected"},

/\*<identifier\_start> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<letter> <identifier\_body>", "<letter> <identifier\_body>", "<letter> <identifier\_body>", "<letter> <identifier\_body>"},

/\*<identifier\_body> \*/{"", "E\_; is expected", "", "", "", "E\_; is expected", "lamda", "lamda", "lamda", "", "E\_( is an invalid character", "lamda", "lamda", "lamda", "lamda", "lamda", "lamda", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<digit> <identifier\_body>", "<letter> <identifier\_body>", "<letter> <identifier\_body>", "<letter> <identifier\_body>", "<letter> <identifier\_body>"},

/\*<dec-list> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<dec\_enter> : <type> ;", "<dec\_enter> : <type> ;", "<dec\_enter> : <type> ;", "<dec\_enter> : <type> ;"},

/\*<dec\_enter> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<identifier\_start> <dec>", "<identifier\_start> <dec>", "<identifier\_start> <dec>", "<identifier\_start> <dec>"},

/\*<dec> \*/{"", "", "", "", "", "", "", "lamda", ", <dec\_enter>", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*<type> \*/{"E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "integer", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected", "E\_integer is expected"},

/\*<stat-list\_enter> \*/{"", "", "", "lamda", "", "<stat> <stat-list>", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<stat> <stat-list>", "<stat> <stat-list>", "<stat> <stat-list>", "<stat> <stat-list>"},

/\*<stat-list> \*/{"", "", "", "lamda", "", "<stat> <stat-list\_enter>", "", "", "", "", "E\_display is expected", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<stat> <stat-list\_enter>", "<stat> <stat-list\_enter>", "<stat> <stat-list\_enter>", "<stat> <stat-list\_enter>"},

/\*<stat> \*/{"", "", "", "", "", "<write\_enter>", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<assign>", "<assign>", "<assign>", "<assign>"},

/\*<write\_enter> \*/{"", "", "", "", "", "display ( <write>", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*<write> \*/{"", "", "", "", "", "", "", "", "E\_unexpected ,", "\"value=\" , <identifier\_start> ) ;", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<identifier\_start> ) ;", "<identifier\_start> ) ;", "<identifier\_start> ) ;", "<identifier\_start> ) ;"},

/\*<assign> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "<identifier\_start> = <expr\_term\_factor\_enter> ;", "<identifier\_start> = <expr\_term\_factor\_enter> ;", "<identifier\_start> = <expr\_term\_factor\_enter> ;", "<identifier\_start> = <expr\_term\_factor\_enter> ;"},

/\*<expr\_term\_factor\_enter>\*/{"", "", "", "", "", "", "", "", "", "", "<term\_enter> <expr>", "", "", "<term\_enter> <expr>", "<term\_enter> <expr>", "", "", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>", "<term\_enter> <expr>"},

/\*<expr> \*/{"", "", "", "", "", "", "lamda", "", "", "", "", "lamda", "", "+ <term\_enter> <expr>", "- <term\_enter> <expr>", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*<term\_enter> \*/{"", "", "", "", "", "", "", "", "", "", "<factor> <term>", "", "", "<factor> <term>", "<factor> <term>", "", "", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>", "<factor> <term>"},

/\*<term> \*/{"", "", "", "", "", "E\_; is expected", "lamda", "", "", "", "", "lamda", "", "lamda", "lamda", "\* <factor> <term>", "/ <factor <term>", "", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*<factor> \*/{"", "", "", "", "", "", "", "", "", "", "( <expr\_term\_factor\_enter> )", "", "", "<number\_start>", "<number\_start>", "", "", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<number\_start>", "<identifier\_start>", "<identifier\_start>", "<identifier\_start>", "<identifier\_start>"},

/\*<number\_start> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "<sign> <digit> <number\_body>", "<sign> <digit> <number\_body>", "", "", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "", "", "", ""},

/\*<number\_body> \*/{"", "", "", "", "", "", "lamda", "", "", "", "", "lamda", "", "lamda", "lamda", "lamda", "lamda", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "<digit> <number\_body>", "E\_can't use letters in number", "E\_can't use letters in number", "E\_can't use letters in number", "E\_can't use letters in number"},

/\*<sign> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "+", "-", "", "", "lamda", "lamda", "lamda", "lamda", "lamda", "lamda", "lamda", "lamda", "lamda", "lamda", "", "", "", ""},

/\*<digit> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "", "", "", ""},

/\*<letter> \*/{"", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "p", "q", "r", "s"}

};

// All the terminals of our language, its used for index searching later

vector<string> terminals

{

"program", "var", "begin", "end.", "integer", "display", ";", ":", ",", "\"value=\"", "(", ")", "=", "+", "-", "\*", "/", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "p", "q", "r", "s"

};

// All the Non-Terminals of our language, its used for index searching later

vector<string> nonTerminals

{

"<prog>", "<identifier\_start>", "<identifier\_body>", "<dec-list>", "<dec\_enter>", "<dec>", "<type>", "<stat-list\_enter>", "<stat-list>", "<stat>", "<write\_enter>", "<write>", "<assign>", "<expr\_term\_factor\_enter>", "<expr>", "<term\_enter>", "<term>", "<factor>", "<number\_start>", "<number\_body>", "<sign>", "<digit>", "<letter>"

};

// All the reserved words, its used to see if we are looking at an identifier or a reserved word

vector<string> reservedWords

{

"program", "var", "begin", "end.", "integer", "display" , "\"value=\""

};

// Creates classes for our language and compiler

language myLanguage(table, terminals, nonTerminals, reservedWords);

compiler myCompiler(myLanguage, readFileName, maxNumberOfLines);

// Runs our compiler and gets the result

myCompiler.setDebugMode(useDebugMode);

bool result = myCompiler.run();

if(result) { // If it compiled display this message

cout << "File accepted" << endl;

}

if(!result) { // If it failed to compile display this message

cout << orginalFileName << " failed to compile" << endl;

cout << "File NOT accepted" << endl;

myCompiler.flushParsingStack();

}

// If in debug mode then shows you all defined identifiers

if(useDebugMode) {

myCompiler.displayAllDefinedIdentifiers();

cout << "Finished tracing " << readFileName << endl;

cout << "You can check your files or press any key and let the program continue" << endl << flush;

system("PAUSE");

}

return myCompiler.errors;

}

void part3(vector<string> errors, bool useDebugMode = false){

// display all error messages

if(useDebugMode){

cout << LINE1;

cout << "PART 3" << endl;

cout << "Attempting to display all the errors found" << endl;

}

if(!errors.empty()){ // if there are error messages then display them

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

cout << "Error " << errors[i] << endl;

}

} else if(useDebugMode){

cout << "No errors found" << endl;

}

if(useDebugMode){

cout << "Finished displaying all errors" << endl << flush;

system("PAUSE");

}

}

void part4(string orginalFileName ,string readFileName, string writeFileName, string writeFileNameWithOutExtension, bool useDebugMode = false){

// convert our custom programming language into c++

if(useDebugMode){

cout << LINE1;

cout << "PART 4" << endl;

cout << "Attempting to translate the cleaned up version of " << orginalFileName << " in " << readFileName << " into c++ in " << writeFileName << endl;

}

// initializing variables we'll need

bool isInVarBlock = false;

bool isInBeginBlock = false;

bool wordWasWritten = false;

string myLine;

// opening files

ifstream MyReadFile(readFileName);

ofstream MyWriteFile(writeFileName);

// adding c++ header to file

string CppHeader = "#include <iostream>\nusing namespace std;\nint main()\n{\n";

MyWriteFile << CppHeader;

// writing to file

while(getline (MyReadFile, myLine)){ // This lets of check each line of our readfile

stringstream s(myLine); // This will let of check each word in each line of our readfile

string word;

//if we are in the var block of code

if(isInVarBlock){

MyWriteFile << "int ";

while(s >> word){

if(word != ":" && word != "integer") MyWriteFile << word << ' ';

}

isInVarBlock = false;

wordWasWritten = true;

} else if(myLine[0] == 'd'){ // if this line has a display

MyWriteFile << "cout << ";

while(s >> word){

if(word != "display" && word != "(" && word != ")" && word != "," && word != ";") MyWriteFile << word << " << ";

}

MyWriteFile << "endl;";

wordWasWritten = true;

} else{ // handles all other edge cases

while(s >> word){

if(word == "end.")isInBeginBlock = false; // checks if we have left the begin block

if(isInBeginBlock){ // if we are in the begin block

MyWriteFile << word << ' ';

wordWasWritten = true;

}

// Checks if we are entering a block of code

// These are at the end so the words 'var' and 'begin' don't get written

if(word == "var") isInVarBlock = true;

if(word == "begin") isInBeginBlock = true;

}

}

if(wordWasWritten) { // If we actually wrote a line this will and a \n at the end of the line

MyWriteFile << "\n";

}

wordWasWritten = false;

}

// adding c++ footer to file

string CppFooter = "return 0;\n}";

MyWriteFile << CppFooter;

// closing files

MyReadFile.close();

MyWriteFile.close();

cout << orginalFileName << " has been compiled into " << writeFileName << endl;

cout << "Attempting to compile " << writeFileName << " with the following command: ";

cout << "g++ " << writeFileName << endl;

string compileCommand = "g++ " + writeFileName;

string runCommand = ".\\" + writeFileNameWithOutExtension + ".exe";

const char\* compileStr = compileCommand.c\_str();

const char\* runStr = runCommand.c\_str();

system(compileStr);

if(doesFileExist(writeFileNameWithOutExtension + ".exe")){

cout << "Attempting to run compiled file" << endl;

system(runStr);

cout << "Compiled file exited" << endl;

} else{

cout << "Failed to compile " << writeFileName << endl;

cout << "Please try to compile it yourself once this program has exited" << endl;

}

}

// Driver Code

int main()

{

// Variables

string orignialFileName, orignialFileNameWithExtenstion;

string cleanedUpFileName = "finalp2.txt";

string endingCppFileName, endingCppFileNameWithExtenstion;

int numberOfLinesInCleanedUpFile;

bool isGoing = true;

bool isInDebugMode = false;

vector<string> errors;

//Main process

welcomeMessage();

while(isGoing){

// Get user input

orignialFileName = getFileNameFromUser("Please enter the name of the exiting text file you want compiled", ".txt", true);

endingCppFileName = getFileNameFromUser("Please enter the name you want for the final C++ file", ".cpp");

isInDebugMode = getYesOrNoResponse("Enable debug mode for this compiler? ");

orignialFileNameWithExtenstion = orignialFileName + ".txt";

endingCppFileNameWithExtenstion = endingCppFileName + ".cpp";

cout << LINE2;

cout << "Running compiler" << endl;

numberOfLinesInCleanedUpFile = part1(orignialFileNameWithExtenstion, cleanedUpFileName, isInDebugMode);

errors = part2(orignialFileNameWithExtenstion ,cleanedUpFileName, numberOfLinesInCleanedUpFile, isInDebugMode);

part3(errors, isInDebugMode);

if(errors.empty())part4(orignialFileNameWithExtenstion ,cleanedUpFileName, endingCppFileNameWithExtenstion, endingCppFileName, isInDebugMode);

else cout << "Can not translate to C++ due to errors" << endl;

cout << LINE2;

isGoing = getYesOrNoResponse("Do you want to compile another file?");

cout << LINE1;

}

goodByeMessage();

return 0;

}