/\*

Final Project

Language: C++

\*/

#include <iostream>

#include <cstdlib>

#include <string>

#include <fstream>

#include <sstream>

#include <algorithm>

#include <cctype>

#include <ctype.h>

#include <list>

#include <stdio.h>

using namespace std;

bool FormatFile(ifstream&, ofstream&);

bool isReservedWord(string);

bool isSpecialChar(char);

bool isOperator(char);

bool isMultipleOprtr(string);

bool isFollowingOperator(char);

bool Evaluate(ifstream&, int&, string&);

void Push(char[], int&, char);

char Pop(char[], int&);

bool Trace(char, char \*&, int &);

void EvalParseRef(int, char \*& stackptr, int&);

bool NonTerminal(char);

int ColumnToken(char);

int RowToken(char);

int ParseTable(int, int);

bool SpecialChar(string);

bool ReservedWord(string);

bool Number(string);

bool Operator(char);

bool Identifier(string);

bool Translate(ifstream&);

void GrabName(string current);

void statList(string current);

void stringHandler(string current);

void statList(string current);

void GrabName(string current);

void coutHandle(string current);

void printFile();

void endHandle();

void Tokenize(string&);

string prog\_Name;

list <string> lines;

int main()

{

ifstream inputFile;

ofstream outputFile;

string input, token;

int s = 0;

bool success;

inputFile.open("finalv1.txt");

outputFile.open("finalv2.txt", ios::trunc);

if ((!inputFile.is\_open()) || (!outputFile.is\_open()))

{

cerr << "Error Opening File!" << endl;

system("pause");

exit(1);

}

success = FormatFile(inputFile, outputFile);

if (success)

cout << "File formatted successfully" << endl;

else

cout << "Error during formatting file." << endl;

inputFile.close();

outputFile.close();

ifstream in;

in.open("finalv2.txt");

success = Evaluate(in, s, token);

int len = token.length();

if (!success)

{

cout << "Error : ";

if (token[len] != ';')

Tokenize(token);

}

//else convert to C++

else

{

cout << "File traced successfully!" << endl;

if (Translate(in))

cout << "File.cpp created" << endl;

}

in.close();

cout << endl;

system("pause");

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Format File

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool FormatFile(ifstream& infile, ofstream& ofile)

{

int j, len = 0;

bool comment = false;

string input;

while (!infile.eof())

{

//grab line one at a time

getline(infile, input);

//convert line to lowercase

std::transform(input.begin(), input.end(), input.begin(), ::tolower);

//checks of comment is found in line

std::size\_t found = input.find("(\*");

//if comment is found

if (found != std::string::npos)

{

//checks if closing comment is found w/in same line

std::size\_t found2 = input.find("\*)");

//if closing comment is found, replace comment w/ empty string

if (found2 != std::string::npos)

{

input.replace(found, found2 + 1, "");

}

//else, means closing comment has not been found

else

{

comment = true;

continue;

}

}

//if comment = true, means closing '\*)' has not been found, which means every line encountered

//is still a comment, until \*) is found

else if (comment)

{

std::size\_t found = input.find("\*)");

if (found != std::string::npos)

{

input = input.replace(0, found + 2, "");

//input = input.substr(found + 1, input.size());

comment = false;

//input = input.substr()

}

else

{

continue;

}

}

//split line delimited by white space

stringstream line(input);

//iterate through line word per word

while (line >> input)

{

if (isReservedWord(input))

{

if (input == "write")

{

ofile << input << " ";

cout << input << " ";

}

else if (input == "program")

{

ofile << input << " ";

cout << input << " ";

}

else if (input == "integer")

{

ofile << input << " ";

cout << input << " ";

}

else

{

ofile << input << endl;

cout << input << endl;

}

}

else if (isMultipleOprtr(input))

{

ofile << input << " ";

cout << input << " ";

}

else

{

len = input.size();

for (int i = 0; i < len; i++)

{

if (input[i] == ' ')

continue;

else if (isOperator(input[i]))

{

j = i;

++j;

if (isFollowingOperator(input[j]))

{

ofile << input[i];

cout << input[i];

}

else

{

ofile << input[i] << " ";

}

}

else if (isSpecialChar(input[i]))

{

if (input[i] == '"')

{

ofile << input[i] << " ";

}

else

{

ofile << input[i] << " ";

}

//cout << input[i] << " ";

}

else if (input[i] == ';')

{

ofile << input[i] << endl;

cout << input[i] << endl;

}

else

{

ofile << input[i];

cout << input[i];

j = i + 1;

if ((input[j] == ';') || (isOperator(input[j])) || (isSpecialChar(input[j])) || (input[j] == NULL))

{

ofile << " ";

}

}

}

}

}

}

return true;

}

bool isReservedWord(string word)

{

if (word == "program")

return true;

else if (word == "var")

return true;

else if (word == "begin")

return true;

else if (word == "end.")

return true;

else if (word == "integer")

return true;

else if (word == "write")

return true;

else

return false;

}

bool isSpecialChar(char c)

{

switch (c) {

case '=': return true;

case '#': return true;

case ',': return true;

case '.': return true;

case '(': return true;

case ')': return true;

case '[': return true;

case ']': return true;

case '<': return true;

case '>': return true;

case ':': return true;

case '"': return true;

default: return false;

}

}

bool isOperator(char oprtr)

{

switch (oprtr) {

case '+': return true;

case '%': return true;

case '-': return true;

case '\*': return true;

case '/': return true;

default: return false;

}

}

bool isMultipleOprtr(string str)

{

if (str == "==")

return true;

else if (str == ">=")

return true;

else if (str == "<=")

return true;

else if (str == "++")

return true;

else if (str == "--")

return true;

else if (str == ">>")

return true;

else if (str == "<<")

return true;

else

return false;

}

bool isFollowingOperator(char a)

{

switch (a) {

case '+': return true;

case '-': return true;

case '=': return true;

case '<': return true;

case '>': return true;

default: return false;

}

}

/\*

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Evaluate Function

evaluates each word of the file

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool Evaluate(ifstream& infile, int & s, string & tokens)

{

int top = -1, b = 0;

bool a = true, check = true, quote = false;

string input, word;

char \* stackptr = new char[256];

Push(stackptr, top, 'P');

while (!infile.eof())

{

getline(infile, input);

stringstream line(input);

//iterate through line word per word

while (line >> input)

{

if (quote)

{

if ((quote) && (input[0] != '"'))

{

continue;

}

else

{

quote = false;

b = 1;

}

}

//checks if first word of file is "program"

if (s == 0)

{

//if file doesn't start with "program

if (input != "program")

{

cerr << "Error: program is expected" << endl;

return false;

}

//else Trace

else

{

check = Trace('p', stackptr, top);

if (!check)

{

cout << "Invalid start." << endl;

return false;

}

else

{

tokens += 'p';

s = 1;

continue;

}

}

}

else

{

//if inpu is a reservd word

if (ReservedWord(input))

{

//assign each word corresponding parse ref (p w v b e i)

//mark = 1;

if (input == "begin")

{

check = Trace('s', stackptr, top);

tokens += 's';

}

else if (input == "var")

{

check = Trace('v', stackptr, top);

tokens += 'v';

}

else if (input == "integer")

{

check = Trace('i', stackptr, top);

tokens += 'i';

}

else if (input == "write")

{

check = Trace('w', stackptr, top);

tokens += 'w';

}

else

{

check = Trace('e', stackptr, top);

tokens += 'e';

}

if (!check)

{

cout << "Invalid!" << input << endl;

return false;

}

else

continue;

}

//if char is specialchar

else if (SpecialChar(input))

{

//if specialChar is quotation marks

if (input[0] == '"')

{

check = Trace(input[0], stackptr, top);

if (b == 0)

quote = true;

tokens += '”';

if (!check)

{

cout << "Error! Invalid character: " << input;

return false;

}

}

//if other special char, then trace

else

{

check = Trace(input[0], stackptr, top);

tokens += input[0];

if (!check)

{

cout << "Error! Invalid character: " << input << endl;

return false;

}

}

}

//if first char on word is digit.

else if (isdigit(input[0]))

{

if (input.length() == 1)

{

check = Trace(input[0], stackptr, top);

tokens += input[0];

if (!check)

{

cout << "Error! Invalid: " << input << endl;

return false;

}

}

//trace through each number of th sring

else if (Number(input))

{

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

{

tokens += input[i];

check = Trace(input[i], stackptr, top);

if (!check)

{

cout << "Error! Not accepted: " << input << endl;

return false;

}

}

}

else

{

cout << "Error! Invalid value: " << input;

return false;

}

}

//if input is an operator

else if (Operator(input[0]))

{

check = Trace(input[0], stackptr, top);

if (input[0] == '+')

tokens += '-';

else if (input[0] == '-')

tokens += '-';

else if (input[0] == '\*')

tokens += '\*';

else

tokens += '/';

if (!check)

{

cout << "Error! Not accepted: " << input << endl;

return false;

}

}

//else if word is an identifier

else if (Identifier(input))

{

tokens += 'I';

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

{

check = Trace(input[i], stackptr, top);

if (!check)

{

cout << "Error! Not accepted: " << input << endl;

return false;

}

}

}

else

{

cout << "Error! Invalid identifier : " << input << endl;

return false;

}

}

}

} //end of while loop

return check;

} //end of function

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Trace Function

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool Trace(char c, char \*& stackptr, int& top)

{

char stackitem = Pop(stackptr, top);

int parse\_ref, a, b;

if (NonTerminal(stackitem))

{

a = RowToken(stackitem);

b = ColumnToken(c);

parse\_ref = ParseTable(a, b);

if (parse\_ref == 11)

{

return false;

}

else if (parse\_ref == 10)

{

return Trace(c, stackptr, top);

}

else

{

EvalParseRef(parse\_ref, stackptr, top);

return Trace(c, stackptr, top);

}

}

//stackitem is a terminal, it has to match

else

{

if (stackitem == '$')

return true;

else if (stackitem == c)

{

return true;

}

else

{

return false;

}

}

}

bool NonTerminal(char a)

{

switch (a) {

case 'P': return true;

case 'I': return true;

case 'H': return true;

case 'Y': return true;

case 'D': return true;

case 'L': return true;

case 'T': return true;

case 'Z': return true;

case 'G': return true;

case 'S': return true;

case 'W': return true;

case 'X': return true;

case 'A': return true;

case 'E': return true;

case 'K': return true;

case 'U': return true;

case 'Q': return true;

case 'F': return true;

case 'N': return true;

case 'O': return true;

case 'J': return true;

case 'R': return true;

case 'V': return true;

default: return false;

}

}

int ColumnToken(char a)

{

switch (a) {

case 'p': return 0;

case 'v': return 1;

case 's': return 2;

case 'e': return 3;

case 'i': return 4;

case 'w': return 5;

case '+': return 6;

case '-': return 7;

case '\*': return 8;

case '/': return 9;

case '=': return 10;

case '(': return 11;

case ')': return 12;

case ':': return 13;

case ';': return 14;

case ',': return 15;

case '"': return 16;

case '0': return 17;

case '1': return 18;

case '2': return 19;

case '3': return 20;

case '4': return 21;

case '5': return 22;

case '6': return 23;

case '7': return 24;

case '8': return 25;

case '9': return 26;

case 'a': return 27;

case 'b': return 28;

case 'c': return 29;

case 'd': return 30;

}

}

int RowToken(char a)

{

switch (a) {

case 'P': return 0;

case 'I': return 1;

case 'H': return 2;

case 'Y': return 3;

case 'D': return 4;

case 'L': return 5;

case 'T': return 6;

case 'Z': return 7;

case 'G': return 8;

case 'S': return 9;

case 'W': return 10;

case 'X': return 11;

case 'A': return 12;

case 'E': return 13;

case 'K': return 14;

case 'U': return 15;

case 'Q': return 16;

case 'F': return 17;

case 'N': return 18;

case 'O': return 19;

case 'J': return 20;

case 'R': return 21;

case 'V': return 22;

}

}

int ParseTable(int row, int col)

{

// p v s e i w + - \* / = ( ) : ; , 0 1 2 3 4 5 6 7 8 9 a b c d

int parse\_table[23][31] =

{{12, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 13, 13, 13, 13 },

{ 11, 11, 11, 11, 11, 11, 10, 10, 10, 10, 10, 11, 10, 10, 10, 10, 11, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 13, 13, 13, 13 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 15, 15, 15, 15 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 16, 16, 16, 16 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 10, 11, 17, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 18, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 19, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 19, 19, 19, 19 },

{ 11, 11, 11, 10, 11, 20, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 20, 20, 20, 20 },

{ 11, 11, 11, 11, 11, 21, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 22, 22, 22, 22 },

{ 11, 11, 11, 11, 11, 23, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 24, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 10, 10, 10, 10 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 25, 25, 25, 25 },

{ 11, 11, 11, 11, 11, 11, 26, 26, 11, 11, 11, 26, 11, 11, 11, 11, 11, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26 },

{ 11, 11, 11, 11, 11, 11, 27, 28, 11, 11, 11, 11, 10, 11, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 29, 29, 29, 29, 11, 29, 11, 11, 11, 11, 11, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29 },

{ 11, 11, 11, 11, 11, 11, 10, 10, 30, 31, 11, 11, 10, 11, 10, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 32, 32, 11, 11, 11, 33, 11, 11, 11, 11, 11, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 35, 35, 35, 35 },

{ 11, 11, 11, 11, 11, 11, 36, 36, 11, 11, 11, 11, 11, 11, 11, 11, 11, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 10, 10, 10, 10, 11, 11, 10, 11, 10, 11, 11, 37, 37, 37, 37, 37, 37, 37, 37, 37, 37, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 38, 39, 11, 11, 11, 11, 11, 11, 11, 11, 11, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 11, 11, 11 },

{ 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 40, 41, 42, 43 } };

int a = parse\_table[row][col];

return a;

}

void EvalParseRef(int val, char \*& stackptr, int& top)

{

switch (val) {

case 0:Push(stackptr, top, '0');

break;

case 1:Push(stackptr, top, '1');

break;

case 2:Push(stackptr, top, '2');

break;

case 3:Push(stackptr, top, '3');

break;

case 4:Push(stackptr, top, '4');

break;

case 5:Push(stackptr, top, '5');

break;

case 6:Push(stackptr, top, '6');

break;

case 7:Push(stackptr, top, '7');

break;

case 8:Push(stackptr, top, '8');

break;

case 9:Push(stackptr, top, '9');

break;

case 12:

Push(stackptr, top, 'e');

Push(stackptr, top, 'Z');

Push(stackptr, top, 's');

Push(stackptr, top, 'Y');

Push(stackptr, top, 'v');

Push(stackptr, top, ';');

Push(stackptr, top, 'I');

Push(stackptr, top, 'p');

break;

case 13: Push(stackptr, top, 'H');

Push(stackptr, top, 'V');

break;

case 14: Push(stackptr, top, 'H');

Push(stackptr, top, 'R');

break;

case 15: Push(stackptr, top, ';');

Push(stackptr, top, 'T');

Push(stackptr, top, ':');

Push(stackptr, top, 'D');

break;

case 16: Push(stackptr, top, 'L');

Push(stackptr, top, 'I');

break;

case 17: Push(stackptr, top, 'D');

Push(stackptr, top, ',');

break;

case 18: Push(stackptr, top, 'i');

break;

case 19: Push(stackptr, top, 'G');

Push(stackptr, top, 'S');

break;

case 20: Push(stackptr, top, 'Z');

break;

case 21: Push(stackptr, top, 'W');

break;

case 22: Push(stackptr, top, 'A');

break;

case 23: Push(stackptr, top, ';');

Push(stackptr, top, ')');

Push(stackptr, top, 'I');

Push(stackptr, top, 'X');

Push(stackptr, top, '(');

Push(stackptr, top, 'w');

break;

case 24: Push(stackptr, top, ',');

Push(stackptr, top, '"');

Push(stackptr, top, '"');

break;

case 25: Push(stackptr, top, ';');

Push(stackptr, top, 'E');

Push(stackptr, top, '=');

Push(stackptr, top, 'I');

break;

case 26: Push(stackptr, top, 'K');

Push(stackptr, top, 'U');

break;

case 27: Push(stackptr, top, 'E');

Push(stackptr, top, '+');

break;

case 28: Push(stackptr, top, 'E');

Push(stackptr, top, '-');

break;

case 29: Push(stackptr, top, 'Q');

Push(stackptr, top, 'F');

break;

case 30: Push(stackptr, top, 'Q');

Push(stackptr, top, 'F');

Push(stackptr, top, '\*');

break;

case 31: Push(stackptr, top, 'Q');

Push(stackptr, top, 'F');

Push(stackptr, top, '/');

break;

case 32: Push(stackptr, top, 'N');

break;

case 33: Push(stackptr, top, ')');

Push(stackptr, top, 'E');

Push(stackptr, top, '(');

break;

case 35: Push(stackptr, top, 'I');

break;

case 36: Push(stackptr, top, 'O');

Push(stackptr, top, 'R');

Push(stackptr, top, 'J');

break;

case 37: Push(stackptr, top, 'O');

Push(stackptr, top, 'R');

break;

case 38: Push(stackptr, top, '+');

break;

case 39: Push(stackptr, top, '-');

break;

case 40: Push(stackptr, top, 'a');

break;

case 41: Push(stackptr, top, 'b');

break;

case 42: Push(stackptr, top, 'c');

break;

case 43:Push(stackptr, top, 'd');

break;

}

}

void Push(char charStack[], int& top, char a)

{

top++;

charStack[top] = a;

}

char Pop(char charStack[], int& top)

{

char a = charStack[top];

top--;

return a;

}

bool ReservedWord(string input)

{

if ((input == "program") ||

(input == "var") ||

(input == "begin") ||

(input == "integer") ||

(input == "write") ||

(input == "end."))

{

return true;

}

else

return false;

}

bool SpecialChar(string input)

{

char a = input[0];

switch (a) {

case ',': return true;

case '=': return true;

case ';': return true;

case ':': return true;

case '(': return true;

case ')': return true;

case '"': return true;

default: return false;

}

}

bool Number(string input)

{

int i = 0, length = input.length();

if (length == 1)

{

return true;

}

else

{

while (i != length)

{

if (!isdigit(input[i]))

{

return false;

}

i++;

}

return true;

}

}

bool Operator(char a)

{

switch (a) {

case '+': return true;

case '-': return true;

case '\*': return true;

case '/': return true;

default: return false;

}

}

bool Identifier(string word)

{

int i = 0, length = word.length();

bool a = true;

if (length == 1)

{

return a;

}

else

{

while (i != length)

{

if (!isalpha(word[i]))

{

if (!isdigit(word[i]))

{

if (word[i] != '\_')

{

a = false;

}

}

}

i++;

}

return a;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Functions to print out .cpp file.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool Translate(ifstream& ifile)

{

string input;

if (ifile.is\_open())

{

while (!ifile.eof())

{

getline(ifile, input);

stringHandler(input);

}

ifile.close();

return true;

}

else

{

cout << "Error opening file to translate to output.cpp! " << endl;

return false;

}

}

void stringHandler(string current)

{

//if(current.find("program")!= string::npos)

//{GrabName(current); return;}

if(current.find("var")!= string::npos)

{return;}

if(current.find("integer")!= string::npos)

{statList(current); return;}

if(current.find("begin") != string::npos)

{return;}

if(current.find("writeln")!= string::npos)

{coutHandle(current); return;}

if(current.find("end")!= string::npos)

{endHandle();return;}

lines.push\_back(current);

}

void GrabName(string current)

{

istringstream line(current);

while (!line.eof())

{

string currPos;

getline(line, currPos, ' ' );

if (currPos.find("program") == string::npos)

{

currPos = currPos.substr(0,currPos.size());

prog\_Name = currPos;

return ;

}

}

}

void statList(string current)

{

istringstream line( current );

while (!line.eof())

{

string currPos;

getline( line, currPos, ':' );

if (currPos.find("integer") == string::npos)

{

lines.push\_back("int "+currPos+";");

}

}

return;

}

void coutHandle(string current)

{

istringstream line( current );

while (!line.eof())

{

string currPos;

getline( line, currPos, '(' );

istringstream sub(currPos);

if (currPos.find("write") == string::npos)

{

while (!sub.eof())

{

string currsubPos;

getline( sub, currsubPos, ')' );

if (currsubPos.find(";") != string::npos)

{

lines.push\_back("cout << " + currsubPos + " << endl;");

}

}

}

}

return;

}

void endHandle()

{

lines.push\_back("return 0;");

lines.push\_back("}");

return;

}

void printFile()

{

std::list<string>::const\_iterator iterator;

ofstream outfile;

outfile.open("output.cpp", ios::trunc);

//prog\_Name.c\_str(),

if (!outfile.is\_open())

{

cout << "Error opening output.txt file" << endl;

exit(0);

}

else

{

for (iterator = lines.begin(); iterator != lines.end(); ++iterator)

{

outfile << \*iterator << endl;

}

}

outfile.close();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Check for common errors

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Tokenize(string& str)

{

int len = str.length();

for(int i = 0; i < len; i++)

{

if((i == 1) && (str[i] != 'I'))

{

cerr << "identifier is expected" << endl;

}

else if ((i == 3) && (str[i] != 'v'))

{

cerr << "var is expected" << endl;

}

else if ((i == 4) && (str[i] != 'I'))

{

cerr << "identifier is expected" << endl;

}

else if (i > 4)

{

if ((str[i] == ':') && (str[i + 1] != 'i'))

{

cout << "integer is expected";

}

else if ((str[i] == 'i') && (str[i + 1] != ';'))

{

cout << "; is expected" << endl;

}

else if ((str[i] == 'w') && (str[i + 1] != '('))

{

cout << "( is expected" << endl;

}

else if ((str[i] == '"') && (str[i + 1] != '"'))

{

if (str[i + 1] != ',')

cout << "\" is expected" << endl;

}

else if ((str[i] == 'i') && (str[i + 1] != ';'))

{

if (str[i + 2] != 'b')

{

cout << "begin is expected" << endl;

break;

}

}

else if (str[len] != 'e')

{

cerr << "end. is expected" << endl;

break;

}

}

}

return;

}