МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра программной инженерии

Отчет   
по лабораторным работам  
по дисциплине «Проектная деятельность»

Выполнил: Шорин В.Д. Шифр: 171406  
ИПАИТ  
Направление: 09.03.04 «Программная инженерия»  
Группа: 71-ПГ  
Проверил:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Отметка о зачете:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата «\_\_\_\_» \_\_\_\_\_\_ 2019 г.

Орел, 2019 г.

Лабораторная работа 1

«Структура компилятора»

Задание

Изучить структуру lex и yacc.

Теория

Программа Lex имеет следующий вид:

Объявления

%%

Правила трансляции

%%

Вспомогательные функции

Раздел объявлений включает объявления переменных, именованные константы и регулярные определения (последовательность регулярных выражений).

Правила трансляции имеют вид

Шаблон {Действие}

Каждый шаблон является регулярным выражением, которое может использовать регулярные определения из раздела объявлений. Действия представляют собой фрагменты кода, обычно написанные на языке С, хотя созданы многие разновидности Lex для других языков программирования.

Третий раздел содержит различные дополнительные функции, используемые в действиях. Эти функции могут также компилироваться отдельно и загружаться лексическим анализатором во время работы.

Исходная Yacc-программа состоит из трех частей:

Объявления

%%

Правила трансляции

%%

С-подпрограммы поддержки

Часть объявлений

В Yacc-программе имеется два необязательных раздела объявлений. В первом размещаются обычные объявления С, ограниченные %. Здесь мы помещаем объявления временных переменных, используемых правилами трансляции или процедурами второй и третьей частей.

Часть правил трансляции

В этой части спецификации Yacc после первой пары %% мы размещаем правила трансляции. Каждое правило состоит из продукции грамматики и связанных семантических действий.

Часть С-подпрограмм поддержки

Третья часть содержит С-подпрограммы поддержки. Среди них обязательно должна находиться функция лексического анализатора yylex(). Другие функции, такие как подпрограммы восстановления после ошибок, могут быть добавлены при необходимости.

Лексический анализатор yylex() считывает входной поток по одному символу с использованием функции getchar(). Если считанный символ – цифра, ее значение сохраняется в переменной yylval.

Лабораторная работа 2

«Распознавание лексем»

Задание

Определить введенную лексему на принадлежность классам int, float, char.

Теория

Лексема представляет собой последовательность символов исходной программы, которая соответствует шаблону токена и идентифицируется лексическим анализатором как экземпляр токена.

Описание

Программа считывает лексемы из файла, определяет их на принадлежность классам int, float, char и выводит в консоль класс считанной лексемы.

Код

#include <iostream>#include <fstream>#include <regex>#include <vector>#include <string>#include <stack>std::ifstream fin;  
std::string yytext;  
int lex\_type;void yylex() {  
 std::regex int\_reg("-?[1-9][0-9]\*");  
 std::regex float\_reg("-?[0-9]\*\\.[0-9]\*(e-?[1-9][0-9]\*)?");  
 std::regex id\_reg("\_?[a-zA-Z][a-zA-Z0-9]\*");   
 std::string str;  
 if (!fin.eof()) {  
 fin >> str;  
 if (std::regex\_match(str, int\_reg)) {

std::cout << "INT" << std::endl; lex\_type = 1; yytext = str;  
 } else if (std::regex\_match(str, float\_reg)) {

lex\_type = 1; std::cout << "FLOAT" << std::endl;

yytext = str;  
 } else if (std::regex\_match(str, id\_reg)) {

std::cout << "ID" << std::endl;

yytext = str;  
 } else {

std::cout << "UNDEFINED" << std::endl;

yytext = "";

}  
 str = "";  
 }  
}

int main() {  
 fin.open(R"(C:\Users\vscho\Desktop\Translator\Lab1-C++\text.txt)");  
 while (!fin.eof()) {

yylex();

}  
 fin.close();  
 return 0;  
}

Лабораторная работа 3

«Построение списка токенов»

Задание

На основе лабораторной работы 2 сформировать список токенов.

Теория

Токен – представляет собой пару «имя\_токена, значение\_атрибута». «Имя\_токена» представляет собой абстрактный символ, использующийся во время синтаксического анализа. «Значение\_атрибута» указывает на запись в таблице символов, соответствующую данному токену.

Описание

На основании кода лабораторной работы 2, программа формирует список токенов в зависимости от класса, назначенного лексеме. Структура токена состоит из его типа и значения (число или оператор или скобка). Метод определения класса токена из лабораторной работы 2 был модифицирован для определения скобок и операций.

Код

union token\_values {

float f;

char o\_h;

};  
struct Token {

int type;

token\_values value;

};

std::vector<Token> \*tokens = new std::vector<Token>;

void yylex() {  
 std::regex int\_reg("-?[1-9][0-9]\*");  
 std::regex float\_reg("-?[0-9]\*\\.[0-9]\*(e-?[1-9][0-9]\*)?");  
 std::regex id\_reg("\_?[a-zA-Z][a-zA-Z0-9]\*");  
 std::regex operations\_reg (R"(\+|\-|\\*|\/)");  
 std::regex hooks\_reg(R"(\(|\))");  
 std::string str;  
 if (!fin.eof()) {  
 fin >> str;  
 if (std::regex\_match(str, int\_reg)) {

std::cout << "INT" << std::endl;

lex\_type = 1;

yytext = str;  
 } else if (std::regex\_match(str, float\_reg)) {

lex\_type = 1;

std::cout << "FLOAT" << std::endl;

yytext = str;  
 } else if (std::regex\_match(str, id\_reg)) {

std::cout << "ID" << std::endl;

yytext = str;  
 } else if (std::regex\_match(str, operations\_reg)) {  
 lex\_type = 2;

std::cout << "OPERATION" << std::endl;

yytext = str;  
 } else if (std::regex\_match(str, hooks\_reg)) {

lex\_type = 3;

std::cout << "HOOK" << std::endl;

yytext = str;  
 } else {

std::cout << "UNDEFINED" << std::endl;

yytext = "";  
 }  
 str = "";  
 }  
}

void AddToken(int type, token\_values value) {  
 Token new\_token;  
 new\_token.type = type;  
 switch (new\_token.type) {  
 case 1: {

new\_token.value.f = value.f;

break;

}  
 case 2: {

new\_token.value.o\_h = value.o\_h;

break;

}  
 case 3: {

new\_token.value.o\_h = value.o\_h;

break;

}  
 default:

break;  
 }  
 tokens->push\_back(new\_token);  
}  
int main() {  
 fin.open(R"(C:\Users\vscho\Desktop\Translator\Lab1-C++\text.txt)");  
 std::string expression;  
  
 while (!fin.eof()) {  
 yylex();  
 expression += yytext;  
 token\_values new\_value;  
  
 if (lex\_type == 1) {  
 std::string::size\_type sz;  
 new\_value.f = std::stof(yytext, &sz);  
 } else if (lex\_type == 2 || lex\_type == 3) {  
 new\_value.o\_h = yytext.c\_str()[0];  
 }  
  
 AddToken(lex\_type, new\_value);  
 }  
  
 fin.close();  
 return 0;  
}

Лабораторная работа 4

«Разбор выражения»

Задание

Сформировать дерево выражения и вывести его.

Описание

На основе составленного списка токенов функция createPostfixForm формирует список токенов, приведенный к постфиксной форме. Функция calcPriority определяет приоритет операции для правильного построения постфиксной формы. Затем, функция CreateTree, проходя по этому списку, формирует дерево выражения

Код

int calcPriority(char operation) {  
 switch (operation) {  
 case '(': return 1;  
 case '-': return 2;  
 case '+': return 3;  
 case '/': return 4;  
 case '\*': return 5;  
 default: return 0;  
 }  
}  
  
std::vector<Token> createPostfixForm() {  
 std::stack<Token> operations;  
 std::string resultString;  
 std::vector<Token> resTokens;  
  
 for (int i = 0; i < tokens->size(); i++) {  
 if (tokens->at(i).value.o\_h == '+' || tokens->at(i).value.o\_h == '-' || tokens->at(i).value.o\_h == '\*' ||  
 tokens->at(i).value.o\_h == '/') {  
 if (operations.empty()) {  
 operations.push(tokens->at(i));  
 }  
 else {  
 if (operations.top().value.o\_h == '(') {  
 operations.push(tokens->at(i));  
 }  
 else if (calcPriority(tokens->at(i).value.o\_h) > calcPriority(operations.top().value.o\_h)) {  
 operations.push(tokens->at(i));  
 }  
 else {  
 while (calcPriority(tokens->at(i).value.o\_h) <= calcPriority(operations.top().value.o\_h) && operations.top().value.o\_h != '(' && !operations.empty()) {  
 resultString += operations.top().value.o\_h;  
 resultString += " ";  
 resTokens.push\_back(operations.top());  
 operations.pop();  
 if (operations.empty() || operations.top().value.o\_h == '(')  
 {  
 break;  
 }  
 }  
 operations.push(tokens->at(i));  
 }  
 }  
 } else if (tokens->at(i).value.o\_h == '(') {  
 operations.push(tokens->at(i));  
  
 } else if (tokens->at(i).value.o\_h == ')') {  
 while (operations.top().value.o\_h != '(') {  
 resultString += operations.top().value.o\_h;  
 resultString += " ";  
 resTokens.push\_back(operations.top());  
 operations.pop();  
 }  
 operations.pop();  
 } else {  
 resultString += std::to\_string(tokens->at(i).value.f) + " ";  
 resTokens.push\_back(tokens->at(i));  
 }  
 }  
  
 while (!operations.empty()) {  
 resultString += operations.top().value.o\_h;  
 resultString += " ";  
 resTokens.push\_back(operations.top());  
 operations.pop();  
 }  
  
 return resTokens;  
}

Node \*CreateTree(std::vector<Token> tokens) {  
 std::stack<Node\*> nodes;  
 Node \*node, \*node1, \*node2;  
  
 for (int i = 0; i < tokens.size(); ++i) {  
 if (tokens[i].type != 2) {  
 node = new Node(tokens[i]);  
 nodes.push(node);  
 } else {  
 node = new Node(tokens[i]);  
  
 node1 = nodes.top();  
 nodes.pop();  
 node2 = nodes.top();  
 nodes.pop();  
  
 node->right = node1;  
 node->left = node2;  
  
 nodes.push(node);  
 }  
 }  
  
 node = nodes.top();  
 nodes.pop();  
  
 return node;  
}  
  
void PrintTree(Node \*node, int tab) {  
 if (node) {  
 PrintTree(node->left, tab + 1);  
 for (int i = 0; i < tab; ++i) {  
 std::cout << " ";  
 }  
 if (node->token.type == 2) {  
 std::cout << node->token.value.o\_h << std::endl;  
 } else {  
 std::cout << node->token.value.f << std::endl;  
 }  
 PrintTree(node->right, tab + 1);  
 }  
}

int main() {  
 fin.open(R"(C:\Users\vscho\Desktop\Translator\Lab1-C++\text.txt)");  
  
 std::string expression;  
  
 while (!fin.eof()) {  
 yylex();  
 expression += yytext;  
 token\_values new\_value;  
  
 if (lex\_type == 1) {  
 std::string::size\_type sz;  
 new\_value.f = std::stof(yytext, &sz);  
 } else if (lex\_type == 2 || lex\_type == 3) {  
 new\_value.o\_h = yytext.c\_str()[0];  
 }  
  
 AddToken(lex\_type, new\_value);  
 }  
  
 Node \*root = CreateTree(createPostfixForm());  
 PrintTree(root, 0);  
  
 fin.close();  
 return 0;  
}

Лабораторная работа 5

«Вычисление арифметических выражений»

Задание

На основании лабораторной работы 2 вычислить значение выражения по дереву.

Описание

Функция CalculateValue проходит по дереву, построенному в предыдущей лабораторной работе, считывает значения и операции, которые необходимо с ними произвести, выполняет их и возвращает результат.

Код

float CalculateValue(Node \*node){  
 float valueLeft;  
 float valueRight;  
 if(node->left == nullptr && node->right == nullptr ) {

return node->token.value.f;

}  
 valueLeft = CalculateValue(node->left);  
 valueRight = CalculateValue(node->right);  
 switch (node->token.value.o\_h) {  
 case '+': { return valueLeft + valueRight; }  
 case '-': { return valueLeft - valueRight; }  
 case '\*': { return valueLeft \* valueRight; }  
 case '/': { return valueLeft / valueRight; }  
 }  
}

int main() {  
 fin.open(R"(C:\Users\vscho\Desktop\Translator\Lab1-C++\text.txt)");  
 std::string expression;  
  
 while (!fin.eof()) {  
 yylex();  
 expression += yytext;  
 token\_values new\_value;  
  
 if (lex\_type == 1) {

std::string::size\_type sz;

new\_value.f = std::stof(yytext, &sz);  
 } else if (lex\_type == 2 || lex\_type == 3) {

new\_value.o\_h = yytext.c\_str()[0];  
 }  
 AddToken(lex\_type, new\_value);  
 }  
  
 Node \*root = CreateTree(createPostfixForm());  
 PrintTree(root, 0);  
 std::cout << std::endl << "Result: " << CalculateValue(root) << std::endl;  
  
 fin.close();  
 return 0;  
}

Лабораторная работа 6

«Разбор команд языка Паскаль»

Задание

Написать синтаксический анализатор, определяющий верность кода языка Паскаль.

Описание

Функция parse рекурсивно считывает из файла лексемы, функцией lexType определяет их тип (операция или конец операции) и сравнивает с предыдущими считанными лексемами. Если они соответствуют синтаксису, то возвращается true, иначе false.

Код

std::string lexType(std::string lex){  
 if(lex == "begin" || lex == "if" || lex == "while" || lex == "do" || lex == "then" || lex == "else" || lex == ":=") {  
 return "operation";  
 } else if (lex == "end" || lex == ";" || lex == "." || lex == "end." || lex == "end;") {

return "endOperation";  
 } else { return "undefined"; }  
}  
bool parse(std::ifstream &fin, std::string op) {  
 bool isCorrect = false; std::string lex;  
 while (!fin.eof()) {  
 fin>> lex;  
 if (lexType(lex) == "operation") {

isCorrect = parse(fin, lex);  
 } else if (lexType(lex) == "endOperation") {

return isCorrect;  
 } else {  
 if (op == "if" && lex == "BE") {  
 fin>> lex;  
 if (lex == "then") { isCorrect = parse(fin, lex); } else { return false; }  
 } else if (op == "while" && lex == "BE") {  
 fin>> lex;  
 if (lex == "do") { isCorrect = parse(fin, lex);  
 } else { return false; }  
 } else if (op == ":=" && lex == "E") { isCorrect = true;  
 } else if (lex == "ID") { isCorrect = parse(fin, lex);  
 } else { return false; }  
 }  
 if (!isCorrect) return false;  
 }  
 return isCorrect;  
}

int main() {  
 fin.open(R"(C:\Users\vscho\Desktop\Translator\Lab1-C++\text.txt)");  
 std::cout << "Is correct: " << parse(fin, "") << std::endl;  
 fin.close();  
 return 0;  
}

Лабораторная работа 7

Задание

Сформировать дерево программы на языке Паскаль.

Описание

Функция LexTree рекурсивно считывает лексемы из файла, определяет их тип и в зависимости от типа, выполняет соответствующие проверки (например, если встречается if, то далее идет проверка на булевское выражение и then) и добавляет новый узел с соответствующими потомками.

Код

#include <iostream>#include <fstream>#include <vector>

enum lexType { \_assume\_, \_if\_, \_while\_, \_begin\_, \_id\_, \_be\_, \_e\_,};

class LexNode {

public:

lexType type;

std::string value;

std::vector<LexNode\*> operations;

LexNode(){}

LexNode(lexType type, std::string value) {

this->type = type;

this->value = value;

switch (type) {

case \_id\_: break;

case \_be\_: break;

case \_e\_: break;

case \_assume\_: break;

case \_if\_: break;

case \_while\_: break;

case \_begin\_: break;

default: break;

}

}

};

static lexType LexType (std::string lex) {

if (lex == "ID") { return \_id\_;

} else if (lex == "E") { return \_e\_;

} else if (lex == "BE") { return \_be\_;

} else if (lex == ":=") { return \_assume\_;

} else if (lex == "if") { return \_if\_;

} else if (lex == "while") { return \_while\_;

} else if (lex == "begin") { return \_begin\_;

}

}

void LexTree(std::ifstream &fin, std::string prevLex, LexNode\*& root) {

LexNode\* node = nullptr;

std::string lex, operation;

fin >> lex;

lexType type = LexType(lex);

switch (type) {

case \_id\_: LexTree(fin, lex, root); break;

case \_assume\_:

node = new LexNode(type, lex);

node->operations.push\_back(new LexNode(LexType(prevLex), prevLex));

fin >> lex;

//Is E check

node->operations.push\_back(new LexNode(LexType(lex), lex));

fin >> lex;

root->operations.push\_back(node);

break;

case \_if\_:

node = new LexNode(type, lex);

fin >> lex;

//Is BE check

node->operations.push\_back(new LexNode(LexType(lex), lex));

//Is THEN check

fin >> lex;

LexTree(fin, lex, node);

//Is ELSE check

fin >> lex;

LexTree(fin, lex, node);

root->operations.push\_back(node);

break;

case \_while\_:

node = new LexNode(type, lex);

fin >> lex;

//Is BE check

node->operations.push\_back(new LexNode(LexType(lex), lex));

//Is DO check

fin >> lex;

LexTree(fin, lex, node);

root->operations.push\_back(node);

break;

case \_begin\_:

node = new LexNode(type, lex);

std::ios::pos\_type position = fin.tellg();

fin >> lex;

while (!(lex == "end" || lex == "end." || lex == "end;") && !fin.eof()) {

fin.seekg(position);

LexTree(fin, lex, node);

position = fin.tellg();

fin >> lex;

}

if (root == nullptr) { root = node; }

else { root->operations.push\_back(node); }

break;

}

}

void PrintLexTree(LexNode\* node, int offset) {

for (int i = 0; i < offset; i++) { std::cout << " "; }

std::cout << "|->";

std::cout << node->value << std::endl;

offset += node->value.size() + 2;

for (int i = 0; i < node->operations.size(); i++) {

PrintLexTree(node->operations.at(i), offset);

}

}

Лабораторная работа 8

Задание

Соединить дерево языка Паскаль и дерево выражений.

Описание

Модифицированная из предыдущей лабораторной работы функция LexTree помимо того, что считывает лексемы из файла и строит дерево операций, также считывает логические и арифметические выражения и строит их деревья с помощью соответствующих операций CreateTree и CreateBoolTree, являющихся модификацией функции CreateTree из лабораторной работы 4.

Код

#include <iostream>#include <fstream>#include <regex>

#include <vector>#include <string>#include <stack>

int calcPriorityBool(char operation) {

switch (operation) {

case '(': return 1;

case '|': return 2;

case '&': return 3;

case '!': return 4;

default: return 0;

}

}

std::vector<Token> createPostfixFormBool(std::vector<Token> tokens) {

std::stack<Token> operations;

std::string resultString;

std::vector<Token> resTokens;

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

if (tokens.at(i).value.o\_h == '!' || tokens.at(i).value.o\_h == '&' || tokens.at(i).value.o\_h == '|') {

if (operations.empty()) {

operations.push(tokens.at(i));

}

else {

if (operations.top().value.o\_h == '(') {

operations.push(tokens.at(i));

}

else if (calcPriorityBool(tokens.at(i).value.o\_h) > calcPriorityBool(operations.top().value.o\_h)) {

operations.push(tokens.at(i));

}

else {

while (calcPriorityBool(tokens.at(i).value.o\_h) <= calcPriorityBool(operations.top().value.o\_h) && operations.top().value.o\_h != '(' && !operations.empty()) {

resultString += operations.top().value.o\_h;

resultString += " ";

resTokens.push\_back(operations.top());

operations.pop();

if (operations.empty() || operations.top().value.o\_h == '(') { break; }

}

operations.push(tokens.at(i));

}

}

} else if (tokens.at(i).value.o\_h == '(') {

operations.push(tokens.at(i));

}

else if (tokens.at(i).value.o\_h == ')') {

while (operations.top().value.o\_h != '(') {

resultString += operations.top().value.o\_h;

resultString += " ";

resTokens.push\_back(operations.top());

operations.pop();

}

operations.pop();

} else {

resultString += std::to\_string(tokens.at(i).value.f) + " ";

resTokens.push\_back(tokens.at(i));

}

}

while (!operations.empty()) {

resultString += operations.top().value.o\_h;

resultString += " ";

resTokens.push\_back(operations.top());

operations.pop();

}

return resTokens;

}

TokenNode \*CreateBoolTree(std::vector<Token> tokens) {

std::stack<TokenNode\*> nodes;

TokenNode \*node, \*node1, \*node2;

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

if (tokens[i].type == 4) {

node = new TokenNode(tokens[i]);

node1 = nodes.top(); nodes.pop();

node->left = node1;

nodes.push(node);

} else if (tokens[i].type == 5) {

node = new TokenNode(tokens[i]);

nodes.push(node);

}

else {

node = new TokenNode(tokens[i]);

node1 = nodes.top(); nodes.pop();

node2 = nodes.top(); nodes.pop();

node->right = node1; node->left = node2;

nodes.push(node);

}

}

node = nodes.top();

nodes.pop();

return node;

}

int TokenTypeBool(std::string token) {

std::regex operations\_reg (R"(\&|\||\=)");

std::regex hooks\_reg(R"(\(|\))");

if (token == "true" || token == "false") { return 5;

} else if (std::regex\_match(token, operations\_reg)) { return 2;

} else if (std::regex\_match(token, hooks\_reg)) { return 3;

} else if (token == "!") { return 4;

} else return 0;

}

Token StringToTokenBool(std::string tokenString) {

Token token;

int token\_type = TokenTypeBool(tokenString);

if (token\_type == 5) {

if (tokenString == "true") { token.value.f = 1; }

else if (tokenString == "false") { token.value.f = 0; }

}

else if (token\_type == 2 || token\_type == 3 || token\_type == 4) {

token.value.o\_h = tokenString.c\_str()[0];

}

token.type = token\_type;

return token;

}

TokenNode \*GetBoolTokenNode(std::ifstream &fin) {

std::string lex;

std::vector<Token> tokens;

std::ios::pos\_type position;

fin >> lex;

while (true) {

if (lex == "do" || lex == "then") { break; }

position = fin.tellg();

tokens.push\_back(StringToTokenBool(lex));

fin >> lex;

}

fin.seekg(position);

return CreateBoolTree(createPostfixFormBool(tokens));

}

#include <iostream>#include <fstream>#include <vector>

#include "TokenTree.h"#include "BoolExpression.h"

enum lexType { \_assume\_, \_if\_, \_while\_, \_begin\_, \_id\_, \_be\_, \_e\_,};

class LexNode {

public:

lexType type;

std::string value;

std::vector<LexNode\*> operations;

TokenNode\* tokenNode = nullptr;

LexNode(){}

LexNode(lexType type, std::string value) {

this->type = type;

this->value = value;

}

};

static lexType LexType (std::string lex) {

if (lex == "ID") { return \_id\_;

} else if (lex == "E") { return \_e\_;

} else if (lex == "BE") { return \_be\_;

} else if (lex == ":=") { return \_assume\_;

} else if (lex == "if") { return \_if\_;

} else if (lex == "while") { return \_while\_;

} else if (lex == "begin") { return \_begin\_; }

}

void LexTree(std::ifstream &fin, std::string prevLex, LexNode\*& root) {

LexNode\* node = nullptr;

std::string lex, operation;

fin >> lex;

lexType type = LexType(lex);

switch (type) {

case \_id\_: LexTree(fin, lex, root); break;

case \_assume\_: {

node = new LexNode(type, lex);

node->operations.push\_back(new LexNode(LexType(prevLex), prevLex));

//Is E check

node->operations.push\_back(new LexNode(\_e\_, "E"));

std::vector<Token> tokens;

fin >> lex;

while (lex != ";") {

tokens.push\_back(StringToToken(lex));

fin >> lex;

}

node->operations[node->operations.size() - 1]->tokenNode = CreateTree(createPostfixForm(tokens));

root->operations.push\_back(node);

break;

}

case \_if\_:

node = new LexNode(type, lex);

node->operations.push\_back(new LexNode(\_be\_, "BE"));

//Is BE check

node->operations[node->operations.size() - 1]->tokenNode = GetBoolTokenNode (fin);

//Is THEN check

fin >> lex;

LexTree(fin, lex, node);

//Is ELSE check

fin >> lex;

LexTree(fin, lex, node);

root->operations.push\_back(node);

break;

case \_while\_:

node = new LexNode(type, lex);

// node->operations.push\_back(new LexNode(\_while\_, "while"));

//Is BE check

node->operations[node->operations.size() - 1]->tokenNode = GetBoolTokenNode (fin);

//Is DO check

fin >> lex;

LexTree(fin, lex, node);

root->operations.push\_back(node);

break;

case \_begin\_:

node = new LexNode(type, lex);

std::fpos<mbstate\_t> position = fin.tellg();

fin >> lex;

while (!(lex == "end" || lex == "end." || lex == "end;") && !fin.eof()) {

fin.seekg(position);

LexTree(fin, lex, node);

position = fin.tellg();

fin >> lex;

}

if (root == nullptr) {

root = node;

} else {

root->operations.push\_back(node);

}

break;

}

}

void PrintTokenTree(TokenNode\* node, int offset) {

if (node) {

for (int i = 0; i < offset; i++) {

std::cout << " ";

}

std::cout << "|->";

if (node->token.type == 2 || node->token.type == 4) {

std::cout << node->token.value.o\_h << std::endl;

} else if (node->token.type == 5) {

if (node->token.value.f == 0) {

std::cout << "false" << std::endl;

} else if (node->token.value.f == 1) {

std::cout << "true" << std::endl;

}

} else {

std::cout << node->token.value.f << std::endl;

}

offset += 3;

PrintTokenTree(node->left, offset);

PrintTokenTree(node->right, offset);

}

}

void PrintLexTree(LexNode\* node, int offset) {

for (int i = 0; i < offset; i++) { std::cout << " "; }

std::cout << "|->";

std::cout << node->value << std::endl;

offset += node->value.size() + 2;

if (node->tokenNode != nullptr) {

PrintTokenTree(node->tokenNode, offset);

}

for (int i = 0; i < node->operations.size(); i++) {

PrintLexTree(node->operations.at(i), offset);

}

}

#include <iostream>#include <fstream>#include <regex>#include <vector>#include <string>

#include <stack>

union token\_values { float f; char o\_h;};

struct Token { int type; token\_values value;};

class TokenNode {

public:

Token token;

TokenNode \*left = nullptr;

TokenNode \*right = nullptr;

TokenNode(){}

TokenNode(Token t) {

token = t;

}

};

float CalculateValue(TokenNode \*node){

float valueLeft;

float valueRight;

if(node->left == nullptr && node->right == nullptr ) {

return node->token.value.f;

}

valueLeft = CalculateValue(node->left);

valueRight = CalculateValue(node->right);

switch (node->token.value.o\_h)

{

case '+': { return valueLeft + valueRight; }

case '-': { return valueLeft - valueRight; }

case '\*': { return valueLeft \* valueRight; }

case '/': { return valueLeft / valueRight; }

}

}

int calcPriority(char operation) {

switch (operation) {

case '(': return 1;

case '-': return 2;

case '+': return 3;

case '/': return 4;

case '\*': return 5;

default: return 0;

}

}

std::vector<Token> createPostfixForm(std::vector<Token> tokens) {

std::stack<Token> operations;

std::string resultString;

std::vector<Token> resTokens;

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

if (tokens.at(i).value.o\_h == '+' || tokens.at(i).value.o\_h == '-' || tokens.at(i).value.o\_h == '\*' ||

tokens.at(i).value.o\_h == '/') {

if (operations.empty()) {

operations.push(tokens.at(i));

}

else {

if (operations.top().value.o\_h == '(') {

operations.push(tokens.at(i));

}

else if (calcPriority(tokens.at(i).value.o\_h) > calcPriority(operations.top().value.o\_h)) {

operations.push(tokens.at(i));

}

else {

while (calcPriority(tokens.at(i).value.o\_h) <= calcPriority(operations.top().value.o\_h) && operations.top().value.o\_h != '(' && !operations.empty()) {

resultString += operations.top().value.o\_h;

resultString += " ";

resTokens.push\_back(operations.top());

operations.pop();

if (operations.empty() || operations.top().value.o\_h == '(') { break; }

}

operations.push(tokens.at(i));

}

}

} else if (tokens.at(i).value.o\_h == '(') {

operations.push(tokens.at(i));

} else if (tokens.at(i).value.o\_h == ')') {

while (operations.top().value.o\_h != '(') {

resultString += operations.top().value.o\_h;

resultString += " ";

resTokens.push\_back(operations.top());

operations.pop();

}

operations.pop();

} else {

resultString += std::to\_string (tokens.at(i).value.f) + " ";

resTokens.push\_back(tokens.at(i));

}

}

while (!operations.empty()) {

resultString += operations.top().value.o\_h;

resultString += " ";

resTokens.push\_back(operations.top());

operations.pop();

}

return resTokens;

}

TokenNode \*CreateTree(std::vector<Token> tokens) {

std::stack<TokenNode\*> nodes;

TokenNode \*node, \*node1, \*node2;

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

if (tokens[i].type != 2) {

node = new TokenNode(tokens[i]);

nodes.push(node);

} else {

node = new TokenNode(tokens[i]);

node1 = nodes.top(); nodes.pop();

node2 = nodes.top(); nodes.pop();

node->right = node1; node->left = node2;

nodes.push(node);

}

}

node = nodes.top(); nodes.pop();

return node;

}

int TokenType(std::string token) {

std::regex int\_reg("-?[1-9][0-9]\*");

std::regex float\_reg("-?[0-9]\*\\.[0-9]\*(e-?[1-9][0-9]\*)?");

std::regex id\_reg("\_?[a-zA-Z][a-zA-Z0-9]\*");

std::regex operations\_reg (R"(\+|\-|\\*|\/)");

std::regex hooks\_reg(R"(\(|\))");

int lex\_type = 0;

if (std::regex\_match(token, int\_reg)) { lex\_type = 1;

} else if (std::regex\_match(token, float\_reg)) { lex\_type = 1;

} else if (std::regex\_match(token, operations\_reg)) { lex\_type = 2;

} else if (std::regex\_match(token, hooks\_reg)) { lex\_type = 3; }

return lex\_type;

}

Token StringToToken(std::string tokenString) {

Token token;

int token\_type = TokenType(tokenString);

if (token\_type == 1) { std::string::size\_type sz; token.value.f = std::stof(tokenString, &sz);

} else if (token\_type == 2 || token\_type == 3) {

token.value.o\_h = tokenString.c\_str()[0];

}

token.type = token\_type;

return token;

}

int main() {

fin.open(R"(C:\Users\vscho\Desktop\Translator\Lab1-C++\text.txt)", std::ios::binary);

LexNode\* root = nullptr;

LexTree(fin, "", root);

fin.close();

PrintLexTree(root, 0);

return 0;

}