**CIS 675 Hw3-Parser** **Submitted By Lohith Nimmala**

**Input1 (No Error)**

digraph G {

main [shape=box]; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> init [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label="100 times"];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

**Output1**

Input

digraph G {

main [shape=box]; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> init [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label="100 times"];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

Output of lexer(tokens)

1.DIGRAPH 2.ID 3.LBRACE

4.ID 5.LBRACKET 6.ID 7.EQUAL 8.ID 9.RBRACKET 10.SEMICOLON

11.ID 12.EDGEOP 13.ID 14.LBRACKET 15.ID 16.EQUAL 17.ID 18.RBRACKET 19.SEMICOLON

20.ID 21.EDGEOP 22.ID 23.SEMICOLON

24.ID 25.EDGEOP 26.ID 27.LBRACKET 28.ID 29.EQUAL 30.ID 31.RBRACKET 32.SEMICOLON

33.ID 34.EDGEOP 35.ID 36.SEMICOLON

37.ID 38.EDGEOP 39.ID 40.SEMICOLON

41.ID 42.EDGEOP 43.ID 44.SEMICOLON

45.ID 46.EDGEOP 47.ID 48.LBRACKET 49.ID 50.EQUAL 51.ID 52.COMMA 53.ID 54.EQUAL 55.ID 56.RBRACKET 57.SEMICOLON

58.ID 59.LBRACKET 60.ID 61.EQUAL 62.ID 63.RBRACKET 64.SEMICOLON

65.NODE 66.LBRACKET 67.ID 68.EQUAL 69.ID 70.COMMA 71.ID 72.EQUAL 73.ID 74.COMMA 75.ID 76.EQUAL 77.ID 78.RBRACKET 79.SEMICOLON

80.ID 81.EDGEOP 82.ID 83.SEMICOLON

84.RBRACE

Status of Parser execution = Program has no Syntax Errors

**Input2 (missing right brace)**

digraph G {

main [shape=box; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> init [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label="100 times"];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

**Output2**

Input

digraph G {

main [shape=box; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> init [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label="100 times"];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

Output of lexer(tokens)

1.DIGRAPH 2.ID 3.LBRACE

4.ID 5.LBRACKET 6.ID 7.EQUAL 8.ID 9.SEMICOLON

10.ID 11.EDGEOP 12.ID 13.LBRACKET 14.ID 15.EQUAL 16.ID 17.RBRACKET 18.SEMICOLON

19.ID 20.EDGEOP 21.ID 22.SEMICOLON

23.ID 24.EDGEOP 25.ID 26.LBRACKET 27.ID 28.EQUAL 29.ID 30.RBRACKET 31.SEMICOLON

32.ID 33.EDGEOP 34.ID 35.SEMICOLON

36.ID 37.EDGEOP 38.ID 39.SEMICOLON

40.ID 41.EDGEOP 42.ID 43.SEMICOLON

44.ID 45.EDGEOP 46.ID 47.LBRACKET 48.ID 49.EQUAL 50.ID 51.COMMA 52.ID 53.EQUAL 54.ID 55.RBRACKET 56.SEMICOLON

57.ID 58.LBRACKET 59.ID 60.EQUAL 61.ID 62.RBRACKET 63.SEMICOLON

64.NODE 65.LBRACKET 66.ID 67.EQUAL 68.ID 69.COMMA 70.ID 71.EQUAL 72.ID 73.COMMA 74.ID 75.EQUAL 76.ID 77.RBRACKET 78.SEMICOLON

79.ID 80.EDGEOP 81.ID 82.SEMICOLON

83.RBRACE

Missing Right Brace at token9

**Input3 (missing target node)**

digraph G {

main [shape=box]; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label="100 times"];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

**Output3**

Input

digraph G {

main [shape=box]; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label="100 times"];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

Output of lexer(tokens)

1.DIGRAPH 2.ID 3.LBRACE

4.ID 5.LBRACKET 6.ID 7.EQUAL 8.ID 9.RBRACKET 10.SEMICOLON

11.ID 12.EDGEOP 13.ID 14.LBRACKET 15.ID 16.EQUAL 17.ID 18.RBRACKET 19.SEMICOLON

20.ID 21.EDGEOP 22.ID 23.SEMICOLON

24.ID 25.EDGEOP 26.LBRACKET 27.ID 28.EQUAL 29.ID 30.RBRACKET 31.SEMICOLON

32.ID 33.EDGEOP 34.ID 35.SEMICOLON

36.ID 37.EDGEOP 38.ID 39.SEMICOLON

40.ID 41.EDGEOP 42.ID 43.SEMICOLON

44.ID 45.EDGEOP 46.ID 47.LBRACKET 48.ID 49.EQUAL 50.ID 51.COMMA 52.ID 53.EQUAL 54.ID 55.RBRACKET 56.SEMICOLON

57.ID 58.LBRACKET 59.ID 60.EQUAL 61.ID 62.RBRACKET 63.SEMICOLON

64.NODE 65.LBRACKET 66.ID 67.EQUAL 68.ID 69.COMMA 70.ID 71.EQUAL 72.ID 73.COMMA 74.ID 75.EQUAL 76.ID 77.RBRACKET 78.SEMICOLON

79.ID 80.EDGEOP 81.ID 82.SEMICOLON

83.RBRACE

TARGET NODE ID is missing after EDGEOP at 26

**Input4 (missing ID value after in assignment operation)**

digraph G {

main [shape=box]; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> init [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label=];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

**Output4**

Input

digraph G {

main [shape=box]; /\* this is a comment \*/

main -> parse [weight=8];

parse -> execute;

main -> init [style=dotted];

main -> cleanup;

execute -> make\_string;

init -> make\_string;

main -> printf [style=bold,label=];

make\_string [label="make a\nstring"];

node [shape=box,style=filled,color=".7 .3 1.0"];

execute -> compare;

}

Output of lexer(tokens)

1.DIGRAPH 2.ID 3.LBRACE

4.ID 5.LBRACKET 6.ID 7.EQUAL 8.ID 9.RBRACKET 10.SEMICOLON

11.ID 12.EDGEOP 13.ID 14.LBRACKET 15.ID 16.EQUAL 17.ID 18.RBRACKET 19.SEMICOLON

20.ID 21.EDGEOP 22.ID 23.SEMICOLON

24.ID 25.EDGEOP 26.ID 27.LBRACKET 28.ID 29.EQUAL 30.ID 31.RBRACKET 32.SEMICOLON

33.ID 34.EDGEOP 35.ID 36.SEMICOLON

37.ID 38.EDGEOP 39.ID 40.SEMICOLON

41.ID 42.EDGEOP 43.ID 44.SEMICOLON

45.ID 46.EDGEOP 47.ID 48.LBRACKET 49.ID 50.EQUAL 51.ID 52.COMMA 53.ID 54.EQUAL 55.RBRACKET 56.SEMICOLON

57.ID 58.LBRACKET 59.ID 60.EQUAL 61.ID 62.RBRACKET 63.SEMICOLON

64.NODE 65.LBRACKET 66.ID 67.EQUAL 68.ID 69.COMMA 70.ID 71.EQUAL 72.ID 73.COMMA 74.ID 75.EQUAL 76.ID 77.RBRACKET 78.SEMICOLON

79.ID 80.EDGEOP 81.ID 82.SEMICOLON

83.RBRACE

Value is not assigned at token 55

**LexAndParser.java (main file)**

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class LexAndParser {

private static BufferedReader br;

public static void main(String[] args) throws IOException {

lex l1 = new lex();

// Fetching input file

File file = new File("C:\\Users\\lohit\\Desktop\\input.txt");

br = new BufferedReader(new FileReader(file));

String st;

System.out.println("Input");

while ((st = br.readLine()) != null) {

System.out.println(st);

}

int count=1;

// Initiating List variable to capture all the tokens from lexer

Map<Integer,String> Tokens = new HashMap<Integer, String>();

// Printing output content

System.out.println("Output of lexer(tokens)");

br = new BufferedReader(new FileReader(file));

while ((st = br.readLine()) != null) {

st = st.replaceAll("<.\*>", " ID "); // Replacing all the text in between ".." as ID

st = st.replaceAll("\".\*\"", " ID "); // Replacing all the text in between < > as ID

st = st.replaceAll("/\\\*.\*\\\*/", " "); // Replacing all the text in between /\* \*/ as blank since those are

st = st.replaceAll("\t", "");

// comment

List<String> kk = l1.lex(st);

for (String s : kk) {

System.out.printf(count+"."+s+" ");

Tokens.put(count,s); // adding all the tokens received in the Tokens list variable

count += 1;

}

System.out.println();

}

System.out.println();

System.out.println();

Parser p1 = new Parser(Tokens);

String status = p1.v\_parser();

System.out.println("Status of Parser execution = "+status);

}

}

**Parser.java**

import java.util.HashMap;

import java.util.Map;

public class Parser {

Map<Integer,String> pTokens = new HashMap<Integer, String>();

int focus = 1;

String Error = "";

String token = "";

Parser(Map<Integer, String> Tokens){

this.pTokens = Tokens;

}

public String v\_parser() {

v\_graph();

if(Error.compareTo("")==0) {

return "Program has no Syntax Erros";

}

return Error;

}

public void v\_graph() {

//graph : [ strict ] (graph | digraph) [ ID ] '{' stmt\_list '}'

v\_Terminal("STRICT", "Optional");

if(v\_Terminal("GRAPH") || v\_Terminal("DIGRAPH")) {

v\_Terminal("ID");

if (v\_Terminal("LBRACE" )) {

if (v\_stmt\_list()) {

if (v\_Terminal("RBRACE")) {

Error ="";

}else {

Error = "Right Brace is missing at token "+ focus;

}

} else {

}

} else {

Error = "Left Brace is missing at token "+ focus;

}

}

}

//To verify the terminals

public Boolean v\_Terminal(String terminal, String op) {

token = pTokens.get(focus);

//System.out.println("\*"+ token+"\*\*\*"+terminal+"\*");

if (token.compareTo(terminal)==0) {

focus = focus+1;

return true; }

else if (op.compareTo("Optional")==0) {

return true;

} else {

Error = terminal + " is missing at token ="+focus;

return false;

}

}

public Boolean v\_Terminal(String terminal) {

token = pTokens.get(focus);

//System.out.println("\*"+ token+"\*\*\*"+terminal+"\*");

if (token.compareTo(terminal)==0) {

focus = focus+1;

return true;

}

Error = terminal + " is missing at token ="+focus;

return false;

}

public Boolean v\_stmt\_list() {

int cfocus = focus;

token = pTokens.get(focus);

//stmt\_list : [ stmt [ ';' ] stmt\_list ]

if (v\_stmt()) {

focus = focus-1;

if (v\_Terminal("SEMICOLON")) {

cfocus = focus;

if (!(v\_Terminal("RBRACE"))) {

Error="";

focus = cfocus;

v\_stmt\_list();

} else {

focus = cfocus;

return true;

}

} else {

if ((v\_Terminal("RBRACE"))) {

focus=focus-1;

return true;

} else {

Error = "Semi colon is missing at token "+focus;

return false;

}

}

}

return false;

}

public Boolean v\_stmt() {

token = pTokens.get(focus);

int cfocus = focus;

int start = focus;

if (v\_assignment() ) {

if(v\_Terminal("SEMICOLON")) {

//System.out.println("Succesful statement"+start);

return true;

}

else {

Error = "Semi Colon is missing at "+focus;

return false;

}

}

else {

//System.out.println("Unsuccesful statement");

//System.out.println("Entered Node Verification statement");

focus = cfocus;

if (v\_node\_stmt() ) {

if (v\_Terminal("SEMICOLON")) {

//System.out.println("Succesful v\_node\_stmt"+start);

Error="";

return true;

} else {

Error = "Semi Colon is missing at "+focus;

return false;

}

} else {

focus = cfocus;

//System.out.println("Entered attr\_stmt Verification statement");

if(v\_attr\_stmt()) {

//System.out.println("Succesful v\_attr\_stmt"+focus);

if (v\_Terminal("SEMICOLON")) {

// System.out.println("Succesful v\_attr\_stmt()"+start);

Error="";

return true;

} else {

Error = "Semi Colon is missing at "+focus;

return false;

}

}

else {

focus = cfocus;

//System.out.println("Entered Edge\_stmt Verification statement");

if(v\_edge\_stmt()) {

//System.out.println("Succesful v\_edge\_stmt"+focus);

if (v\_Terminal("SEMICOLON")) {

// System.out.println("Succesful Edge statement "+start);

Error="";

return true;

} else {

Error = "Semi Colon is missing at "+focus;

return false;

}

}

else {

focus = cfocus;

System.out.println("Entered Subgraph Verification statement");

if(v\_subgraph()) {

//System.out.println("Succesful v\_subgraph"+focus);

if (v\_Terminal("SEMICOLON")) {

// System.out.println("Succesful Subgraph statement "+start);

Error="";

return true;

} else {

Error = "Semi Colon is missing at "+focus;

return false;

}

} else {

return false;

}

}

}

}

}

}

// ID = ID

public Boolean v\_assignment() {

if (v\_Terminal("ID") ) {

if (v\_Terminal("EQUAL")) {

if (v\_Terminal("ID")) {

//System.out.println("Succesful assignment");

return true;

} else {

Error="Value is not assigned at token "+focus;

System.out.println(Error);

System.exit(0);

return false;

}

} else {

return false;

}

} else {

return false;

}

}

public Boolean v\_subgraph() {

int cfocus = focus;

Boolean prev =false;

if (v\_Terminal("SUBGRAPH")) {

cfocus = focus;

prev = true;

if (v\_Terminal("ID")) {

cfocus =focus;

}

}

Error="";

//if (v\_Terminal("LBRACE" ) && v\_stmt\_list() && v\_Terminal("RBRACE") ) {

if (v\_Terminal("LBRACE" )) {

cfocus=focus;

if (v\_stmt\_list()) {

if (v\_Terminal("RBRACE")){

Error="";

return true;

} else {

System.out.println("RBRACE is missing at token="+focus);

System.exit(0);

return false;

}

} else {

return false;

}

} else {

if (prev) {

System.out.println("LBRACE is missing at token="+focus);

System.exit(0);

return false;

} else

{

return false;

}

}

}

public Boolean v\_node\_stmt () {

int cnfocus = focus;

if ( v\_node\_id()) {

cnfocus = focus;

if (v\_Terminal("LBRACKET")) {

cnfocus=focus;

focus=focus-1;

if (v\_attr\_list()) {

return true;

} else {

focus =cnfocus;

System.out.println(Error+focus);

System.exit(0);

return true;

}

}

}

return false;

}

public Boolean v\_node\_id() {

int cfocus =focus;

if (v\_Terminal("ID")) {

cfocus = focus;

if (v\_Terminal("COLON")) {

focus=focus-1;

return v\_port();

} else {

return true;

}

}

return false;

}

public Boolean v\_port( ) {

int cfocus = focus;

Boolean prev = false;

if(v\_Terminal("COLON")) {

cfocus=focus;

if (v\_Terminal("ID")) {

prev = true;

if(v\_Terminal("COLON")) {

if( v\_compass\_pt()) {

return true;

}

else {

System.out.println("Commpass\_pt is missing at "+focus);

System.exit(0);

return false;

}

}

}

if(v\_compass\_pt()) {

return true;

}

else {

System.out.println("ID or Commpass\_pt is missing at "+focus);

System.exit(0);

return false;

}

}

else {

return false;

}

}

public Boolean v\_compass\_pt() {

if (v\_Terminal("N") || v\_Terminal("NE") || v\_Terminal("E") || v\_Terminal("SE") || v\_Terminal("S") || v\_Terminal("SW") || v\_Terminal("W") || v\_Terminal("NW") || v\_Terminal("C") || v\_Terminal("UNDERSCORE") ) {

Error = "";

return true;

}

return false ;

}

public Boolean v\_attr\_stmt () {

int cfocus= focus;

if (v\_Terminal("GRAPH") || v\_Terminal("NODE") || v\_Terminal("EDGE")) {

Error = "";

cfocus= focus;

if (v\_attr\_list()) {

return true;

} else {

focus = cfocus;

return false;

}

}

return false;

}

public Boolean v\_attr\_list() {

int cfocus= focus;

Boolean prev = false;

//System.out.println("Entered att list "+cfocus);

if (v\_Terminal("LBRACKET")) {

cfocus = focus;

if (v\_a\_list()) {

cfocus =focus;

if (v\_Terminal("RBRACKET")) {

cfocus=focus;

if (v\_Terminal("LBRACKET")) {

focus= cfocus;

v\_attr\_list();

return true;

} else {

return true;

}

} else {

Error = "Right Bracket is missing";

System.out.println("Missing Right Brace at token"+focus);

System.exit(0);

return false;

}

} else {

Error = "Invalid alist";

focus = cfocus;

return false ;

}

}

else {

Error = "LBRACKET is missing at token "+focus;

return false;

}

}

public Boolean v\_a\_list() {

int cfocus = focus;

boolean prev = false;

//ID '=' ID [ (';' | ',') ] [ a\_list ]

if (v\_assignment())

{

cfocus=focus;

// if (v\_Terminal("SEMICOLON") || v\_Terminal("COMMA")){

if (v\_Terminal("COMMA")){

prev= true;

cfocus=focus;

//System.out.println("COMMA");

if (v\_Terminal("ID")) {

focus =focus-1;

cfocus=focus;

if (v\_a\_list()) {

return true;

} else {

focus=cfocus;

return false;

}

}

else {

if (v\_Terminal("RBRACKET")) {

focus = focus-1;

cfocus=focus;

prev=false;

// System.out.println("recevied RBRACKET in a list"+focus);

return true;

} else {

Error = "Right Bracket is missing";

System.out.println("Missing Right Brace at token"+focus);

System.exit(0);

return false;

}

}

} else {

if (v\_Terminal("RBRACKET")) {

focus = focus-1;

cfocus=focus;

prev=false;

// System.out.println("recevied RBRACKET in a list"+focus);

return true;

} else {

Error = "Right Bracket is missing";

System.out.println("Missing Right Brace at token"+focus);

System.exit(0);

return false;

}

}

}

return false;

}

public Boolean v\_edge\_stmt() {

int cfocus=focus;

int start = focus;

Boolean prev = false;

if (v\_node\_id()) {

prev =true;

//System.out.println("Edge Node pass");

}

if (!prev) {

focus=cfocus;

if(v\_subgraph()) {

prev =true;

//System.out.println("Edge Subgraph pass");

}

}

if (prev) {

//System.out.println("Edge Node or Subgraphpass");

if (v\_edgeRHS()) {

prev = true;

cfocus=focus;

if (v\_Terminal("LBRACKET")) {

focus=focus-1;

return v\_attr\_list();

} else {

return true;

}

} else {

focus=start;

return false;

}

}

focus=start;

return false;

}

public Boolean v\_edgeRHS() {

int start=focus;

int cfocus=focus;

Boolean prev= false;

if (v\_Terminal("EDGEOP")) {

cfocus=focus;

if(v\_node\_id()) {

prev=true;

cfocus=focus;

}

if (!prev) {

focus=cfocus;

if(v\_subgraph()) {

prev=true;

cfocus=focus;

}

else {

System.out.println("TARGET NODE ID is missing after EDGEOP at "+cfocus);

System.exit(0);

return false;

}

}

if(prev) {

if(v\_Terminal("EDGEOP")) {

focus=focus-1;

return v\_edgeRHS();

}

else {

return true;

}

}

else {

return false;

}

}

else {

return false;

}

}

}

**Lex.java**

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.regex.Pattern;

public class lex {

static String tok = "";

static int state = 0;

public static List<String> lex(String input) {

tok = "";

state = 0;

List<String> result = new ArrayList<String>();

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

// System.out.println(input.charAt(i));

switch(input.charAt(i)) {

case '{':

if( state == 1) result.add(checkID(tok));

setzero();

i++;

result.add("LBRACE");

break;

case '}':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("RBRACE");

break;

case ']':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("RBRACKET");

break;

case '[':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("LBRACKET");

break;

case '=':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("EQUAL");

break;

case ':':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("COLON");

break;

case ';':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("SEMICOLON");

break;

case ',':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("COMMA");

break;

/\*case '\_':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

result.add("UNDERSCORE");

break;\*/

case ' ':

if( state ==1) result.add(checkID(tok));

setzero();

i++;

break;

default : state = 1;

tok = tok + Character.toString(input.charAt(i));

i++;

}

}

if( state ==1) result.add(checkID(tok));

return result;

}

public static void setzero() {

tok = "";

state = 0;

}

public static String checkID(String tt) {

//System.out.println("Checking-->"+tt);

String[] predefined = {"node", "edge", "graph", "digraph", "subgraph", "strict","id","->","n","ne","e","se","s","sw","w","nw","c" };

for(String s : predefined) {

if (s.toUpperCase().compareTo(tt.toUpperCase())==0) {

if ((s.toUpperCase().compareTo("->"))==0) {

return "EDGEOP";

}

return s.toUpperCase();

}

}

//Checking for ID pattern

//Any string of alphabetic ([a-zA-Z\200-\377]) characters, underscores ('\_') or digits ([0-9]), not beginning with a digit;

if(Pattern.matches("[^0-9]+[a-zA-Z\_0-9\200-\377]?", tt)) {

return "ID";

}

//a numeral [-]?(.[0-9]+ | [0-9]+(.[0-9]\*)? );

else if (Pattern.matches("[-]?(.[0-9]+|[0-9]+(.[0-9]\*)?)",tt)) {

return "ID";

}

//if none of those matched considering it as error

return "ERROR-->"+tt;

}

}