Compiler design

Lab

Gaurav mitra

Roll : 001410501037

BCSE 3RD YEAR 2ND SEMESTER

CODE

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to efficiently save the production rules of the grammar.

\* Each production is saved in an arraylist.

\*

\* \*/

import java.util.ArrayList;

public class Production {

//nonT : Instance variable for storing the left hand side of the production.

//productions : The right hand side of the productions

public String nonT;

ArrayList<String> productions = new ArrayList<>();

//Constructor

Production(String line)throws NullPointerException{

int index = getIndex(line,"->");

nonT = line.substring(0,index).trim();

String rest = line.substring(index + 2,line.length()).trim();

//System.out.println(rest);

String[] prod = rest.split("\\|");

//System.out.println(prod.length);

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

try {

//System.out.println(prod[i].trim());

productions.add(prod[i].trim());

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

}

catch (Exception e) {

continue;

}

}

}

// Overloaded Constructor

public Production() {

}

//Index of pattern

private int getIndex(String string,String pattern) {

return string.indexOf(pattern);

}

// Print all the productions of the grammar

public void printProductions() {

String buffer = new String(nonT + " -> ");

for(String prod : productions) {

buffer += prod + " | ";

}

System.out.println(buffer);

}

}

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to efficiently save the grammar of the language.

\* Also contains the list of the nonTerminals of the grammar.

\*

\* \*/

import java.util.ArrayList;

import java.util.HashMap;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

public class Grammar {

//grammar : Production rules making up the grammar

//nonTerminals : The list of nonTerminals of the grammar.

ArrayList<Production> grammar = new ArrayList<>();

ArrayList<String> nonTerminals = new ArrayList<>();

// Constructor

Grammar() {

}

// Overloaded Constructor

Grammar(String filename)throws IOException,NullPointerException {

File file = new File(filename);

FileReader fr = new FileReader(file);

BufferedReader br = new BufferedReader(fr);

String line = "";

while(true) {

line = br.readLine();

if(line != null) {

Production p = new Production(line);

grammar.add(p);

nonTerminals.add(p.nonT);

}

else

break;

}

br.close();

}

// Add production rule to the grammar. utility function.

public void add(Production p) {

grammar.add(p);

nonTerminals.add(p.nonT);

}

// Print the production rules of the grammar.

public void printGrammar()throws NullPointerException{

for(Production p : grammar) {

p.printProductions();

}

}

//Update the production rules of the grammar.

public void update(Grammar g) {

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

this.grammar.add(g.grammar.get(i));

this.nonTerminals.add(g.grammar.get(i).nonT);

}

}

// Store the productions in a map for later use.

public void storeMap(HashMap<String,Production> lookup) {

for(Production p : grammar) {

lookup.put(p.nonT, p);

}

}

}

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to remove the left recursion from the Grammar

\* given as an input.

\* A new left recursion removed grammar is saved in a file.

\*

\* \*/

import java.io.IOException;

public class LeftRecursion {

//g : Saves the grammar

public Grammar g;

//Constructor

public LeftRecursion(Grammar g) {

this.g = g;

}

// Function to remove left recursion

public void removeLL() {

int nonTlen = g.nonTerminals.size();

Grammar newgrammar = new Grammar();

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

Grammar temp = new Grammar();

Production p = g.grammar.get(i);

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

p = removeIndirect(p,g.grammar.get(j));

temp.add(p);

}

String prettyprinter = p.nonT + " -> ";

for(String prod : p.productions)

prettyprinter += prod + " | ";

System.out.println(prettyprinter);

newgrammar.update(removeDirect(temp));

}

}

// Helper function to remove direct left recursion

public Grammar removeDirect(Grammar gram) {

Grammar temp = new Grammar();

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

Production p = gram.grammar.get(i);

for(int j = 0; j < p.productions.size(); j++) {

if(p.productions.get(j).startsWith(p.nonT)) {

System.out.print("Direct Visited");

for(int k = 0; k < p.productions.size(); k++) {

if(!p.productions.get(k).startsWith(p.nonT)) {

Production p1 = new Production();

p1.nonT = p.nonT + "'";

p1.productions.add(p.productions.get(k) + p.productions.get(j).substring(p.nonT.length(),p.productions.get(k).length()));

}

}

}

}

temp.add(p);

}

return temp;

}

// Helper function to remove indirect left recursion

public Production removeIndirect(Production ai,Production aj) {

Production newp = new Production();

newp.nonT = ai.nonT;

for(String aiprod : ai.productions) {

if(aiprod.startsWith(aj.nonT)) {

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

String addP = aj.productions.get(i);

String s = addP + aiprod.substring(aj.nonT.length(),aiprod.length());

newp.productions.add(s);

}

}

else {

newp.productions.add(aiprod);

}

}

return newp;

}

//Main function to test correctness

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

Grammar g1 = new Grammar("prog\_gram1.txt");

LeftRecursion l = new LeftRecursion(g1);

l.removeLL();

}

}

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to compute the first and follow for the Grammar

\*

\* \*/

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashMap;

import java.util.HashSet;

import java.util.Iterator;

import java.util.Set;

public class FirstFollow {

// lookup : Saves the left recursion free grammar

// first : Saves the first of each non Terminal

// follow : Saves the follow of each non Terminal

// nonTer : Set of non Terminals

// terminal : Set of Terminals

// g1 : Reads left recursion free grammar from intermediate file.

HashMap<String,Production> lookup = new HashMap<>();

HashMap<String,Set<String>> first = new HashMap<String,Set<String>>();

HashMap<String,Set<String>> follow = new HashMap<String,Set<String>>();

Set<String> nonTer;

Set<String> terminal;

Grammar g1;

// Constructor

public FirstFollow() throws NullPointerException, IOException {

g1 = new Grammar("program\_ll.txt");

g1.storeMap(lookup);

nonTer = new HashSet<>(g1.nonTerminals);

File file = new File("terminal.txt");

FileReader fr = new FileReader(file);

BufferedReader br = new BufferedReader(fr);

String line = br.readLine();

ArrayList<String> t = new ArrayList<String>(Arrays.asList(line.split(",")));

terminal = new HashSet<String>(t);

terminal.add(",");

br.close();

}

// Function to compute First of a non Terminal

public void computeFirst(String current) {

//System.out.println(current);

if(first.containsKey(current))

return;

else {

Set<String> forcurrent = new HashSet<String>();

for(int i = 0; i < lookup.get(current).productions.size(); i++) {

Iterator<String> iter = nonTer.iterator();

String p1 = lookup.get(current).productions.get(i);

int p1size = p1.length();

boolean has\_non\_T = false;

while(iter.hasNext()) {

String cur\_nonT = iter.next();

int cur\_nonT\_size = cur\_nonT.length();

if(p1size >= cur\_nonT\_size) {

String start = p1.substring(0,cur\_nonT\_size);

if(start.equalsIgnoreCase(cur\_nonT)) {

if(!first.containsKey(cur\_nonT)) {

computeFirst(cur\_nonT);

}

//System.out.println("Non t :" + cur\_nonT);

forcurrent.addAll(first.get(cur\_nonT));

// System.out.println(current + " -> " + forcurrent.toString());

has\_non\_T = true;

break;

}

}

}

if(!has\_non\_T) {

//System.out.println("NonTermianl : " + current);

iter = terminal.iterator();

while(iter.hasNext()) {

String cur\_T = iter.next();

int cur\_T\_size = cur\_T.length();

if(p1size >= cur\_T\_size) {

String start = p1.substring(0,cur\_T\_size);

if(start.equals(cur\_T)) {

forcurrent.add(cur\_T);

// System.out.println(current + " -> " + forcurrent.toString());

}

}

}

}

}

first.put(current, forcurrent);

}

}

//// Function to compute Follow of a non Terminal

public void computeFollow(String current) {

System.out.println("\n\nCurrent : " + current);

int cur\_l = current.length();

Set<String> curFollow = new HashSet<>();

Iterator<String> iter = nonTer.iterator();

while(iter.hasNext()) {

String curNonTer = iter.next();

System.out.println("\ncurrent Non Terminal :" + curNonTer);

for(int i = 0; i < lookup.get(curNonTer).productions.size();i++) {

String p1 = lookup.get(curNonTer).productions.get(i);

System.out.println("Production : " + p1);

int p1size = p1.length();

if(cur\_l > p1size)

continue;

else {

int position = p1.indexOf(current);

System.out.println("Pos : " + position);

if(position == -1)

continue;

position += cur\_l;

String rest = p1.substring(position).trim();

System.out.println("Rest : " + rest);

if(rest.length() > 0) {

/\*String ter = checkTerminal(rest);

if(ter != null) {

System.out.println("\n\nIt is Terminal" + ter);

curFollow.add(ter.trim());

}

else {

ter = checkNonTerminal(rest);

System.out.println("\n\nIt is NonTerminal" + ter);

if(!follow.containsKey(ter) && ter != null) {

System.out.println("Computing Follow of : " + ter);

computeFollow(ter);

}

if(!first.get(ter).contains("e$"))

curFollow.addAll(first.get(ter));

else {

curFollow.addAll(follow.get(ter));

}

}\*/

String ter = checkNonTerminal(rest);

if(ter != null) {

System.out.println("\n\nIt is NonTerminal" + ter);

if(!follow.containsKey(ter) && ter != null) {

System.out.println("Computing Follow of : " + ter);

computeFollow(ter);

}

if(!first.get(ter).contains("e$"))

curFollow.addAll(first.get(ter));

else {

curFollow.addAll(follow.get(ter));

}

}

else {

ter = checkTerminal(rest);

if(ter != null) {

System.out.println("\n\nIt is Terminal" + ter);

curFollow.add(ter.trim());

}

}

}

}

}

}

follow.put(current, curFollow);

}

// Helper function to check for terminals in a production

public String checkTerminal(String s) {

s = s.trim();

Iterator<String> iter = terminal.iterator();

while(iter.hasNext()) {

String start = iter.next();

if(s.startsWith(start))

return start;

}

return null;

}

// Helper function to check for non terminals in a production

public String checkNonTerminal(String s) {

s = s.trim();

Iterator<String> iter = nonTer.iterator();

while(iter.hasNext()) {

String start = iter.next();

if(s.startsWith(start))

return start;

}

return null;

}

// Parent function to compute first of all Non terminals

public void computeFirstOfGrammar() {

Iterator<String> iter = nonTer.iterator();

while(iter.hasNext()) {

computeFirst(iter.next());

}

}

// Prints first set

public void printFirst() {

Set<String> main\_set = new HashSet<>(Arrays.asList("int","void"));

Set<String> prog\_set = new HashSet<>(Arrays.asList("int","float"));

Iterator<String> iter = nonTer.iterator();

while(iter.hasNext()) {

String cur = iter.next();

if(cur.equals("main\_func")) {

//System.out.print("Visited\n");

first.put(cur, main\_set);

}

if(cur.equals("program")) {

first.put(cur, prog\_set);

}

System.out.println(cur + " -> " + first.get(cur).toString());

}

}

// Prints follow set

public void printFollow() {

System.out.println("\n\n\n\n FOLLOW");

Iterator<String> iter = nonTer.iterator();

while(iter.hasNext()) {

String cur = iter.next();

System.out.println(cur + " -> " + follow.get(cur).toString());

}

}

// Parent function to compute first and follow

public void computeAll() {

computeFirstOfGrammar();

printFirst();

Iterator<String> iter = nonTer.iterator();

while(iter.hasNext()) {

String str = iter.next();

if(!follow.containsKey(str))

computeFollow(str);

}

printFollow();

}

}

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to compute the LL(1) parsing table for the grammar

\*

\* \*/

import java.io.IOException;

import java.util.HashMap;

import java.util.Iterator;

public class ParsingTable {

// ptable : Stores the parsing table as Map<Non Terminal,Map<Terrminal, Production>>

// gram : The grammar

// f : First follow object uses the computed value of first follow of the grammar

HashMap<String,HashMap<String,String>> ptable = new HashMap<>();

HashMap<String,Production> gram;

FirstFollow f;

// Constructor

public ParsingTable() throws NullPointerException, IOException {

f = new FirstFollow();

f.computeAll();

gram = f.lookup;

}

// Computes the parsing table for the grammar

public void compute() {

for(String nonT : f.nonTer) {

HashMap<String, String> cell = new HashMap<>();

for(int i = 0; i < gram.get(nonT).productions.size(); i++) {

String p1 = gram.get(nonT).productions.get(i);

//System.out.println(nonT + "->" + p1);

int p1size = p1.length();

Iterator<String> iter = f.terminal.iterator();

boolean isTer = false;

while(iter.hasNext()) {

String cur\_T = iter.next();

int cur\_T\_size = cur\_T.length();

if(p1size >= cur\_T\_size) {

String start = p1.substring(0,cur\_T\_size);

if(start.equals(cur\_T)) {

cell.put(cur\_T,p1);

isTer = true;

}

}

}

if(!isTer) {

iter = f.nonTer.iterator();

while(iter.hasNext()) {

String cur\_nonT = iter.next();

int cur\_nonT\_size = cur\_nonT.length();

if(p1size >= cur\_nonT\_size) {

String start = p1.substring(0,cur\_nonT\_size);

if(start.equals(cur\_nonT)) {

//System.out.println("Start : " + start);

//System.out.println(f.first.get(start).toString());

for(String fir : f.first.get(start)) {

if(!fir.equals("e$")) {

cell.put(fir, p1);

}

else {

for(String fol : f.follow.get(start))

cell.put(fol, p1);

}

}

}

}

}

}

}

}

System.out.print(ptable.toString());

}

// Test main

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

ParsingTable p = new ParsingTable();

p.compute();

}

}

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to compute the Symbol Table for the Grammar

\*

\* \*/

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.Iterator;

import java.util.Map.Entry;

/\* Class Item :

\* Stores each token taken from the Lexical analyser as

\* a member of this class.

\*

\*/

class Item {

private final String types[] = {"KEYWORD","IDENTIFIER","ARITHMATIC","RELATIONAL","ASSIGN","DELIMITER","NUM"};

String token;

String type;

int xpos;

int ypos;

Item(String line) {

String array[] = line.split("\t");

System.out.println(array[0] + " : " + array[1] + " : " + array[2] + " : " + array[3] + " : " + array[4]);

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

array[i] = array[i].trim();

}

this.token = array[2];

this.type = types[Integer.parseInt(array[0]) - 1];

this.xpos = Integer.parseInt(array[3]);

this.ypos = Integer.parseInt(array[4]);

}

}

/\* Class Symbol :

\* Stores the symbols in a form so that they can be saved

\* in a symbol table.

\*

\*/

class Symbol {

String type;

String name;

int level;

int xpos;

int ypos;

public Symbol(Item cur,int curlevel,Item prev) {

if(cur.type.equals("IDENTIFIER")) {

type = prev.token;

name = cur.token;

level = curlevel;

xpos = cur.xpos;

ypos = cur.ypos;

}

else {

type = "NUM";

name = cur.token;

level = curlevel;

xpos = cur.xpos;

ypos = cur.ypos;

}

}

public Symbol(Item cur,int curlevel,Item prev,HashMap<Integer,ArrayList<Symbol>> symbols) {

if(cur.type.equals("IDENTIFIER")) {

if(symbols.containsKey(curlevel)) {

boolean there = false;

for(int i = 0; i < symbols.get(curlevel).size(); i++) {

System.out.println("LEVEL : " + symbols.get(curlevel).get(i).name);

if(symbols.get(curlevel).get(i).name.equals(cur.token)) {

//This token is already there in the ST

there = true;

}

}

if(!there) {

//ADD TO ST

type = prev.token;

name = cur.token;

level = curlevel;

}

}

else {

type = prev.token;

name = cur.token;

level = curlevel;

//ADD TO ST

}

}

}

}

/\* Class SymbolTable :

\* Stores the symbol table as an arraylist of

\* objects of the class Symbol

\*

\*/

public class SymbolTable {

// symbol : A data structure to save the symbol table

// table : Saves the scope along with each symbol in the table.

// This is used just to increase the effiecieny of computation

// in latter stages.

// id : IDENTIFIERS

// cons : CONSTANTS

ArrayList<Item> symbol = new ArrayList<>();

HashMap<Integer,ArrayList<Symbol>> table;

HashMap<String,ArrayList<Symbol>> id = new HashMap<>();

HashMap<String,Symbol> cons = new HashMap<>();

// Constructor

public SymbolTable(String filename) throws IOException {

File file = new File(filename);

FileReader fr = new FileReader(file);

BufferedReader br = new BufferedReader(fr);

while(true) {

String line = br.readLine();

if(line == null)

break;

else {

Item i = new Item(line);

symbol.add(i);

}

}

br.close();

this.table = new HashMap<>();

}

// Generates the symbol table by reading symbols sequentially from

// the lexical analyzer

public void generate() {

int level = 0;

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

if(symbol.get(i).type.equals("DELIMITER") && symbol.get(i).token.equals("{"))

level++;

if(symbol.get(i).type.equals("DELIMITER") && symbol.get(i).token.equals("}"))

level--;

if(symbol.get(i).type.equals("IDENTIFIER")) {

if(i > 0) {

Symbol s = new Symbol(symbol.get(i), level, symbol.get(i - 1));

ArrayList<Symbol> allSymbols = new ArrayList<>();

if(table.containsKey(level))

allSymbols.addAll(table.get(level));

allSymbols.add(s);

table.put(level, allSymbols);

}

else {

System.out.println("ERROR!");

}

}

if(symbol.get(i).type.equals("NUM")) {

if(i > 0) {

Symbol s = new Symbol(symbol.get(i), level, symbol.get(i - 1));

ArrayList<Symbol> allSymbols = new ArrayList<>();

if(table.containsKey(level))

allSymbols.addAll(table.get(level));

allSymbols.add(s);

table.put(level, allSymbols);

}

else {

System.out.println("ERROR!");

}

}

}

for(int i = 0; i < 6; i++) {

if(table.containsKey(i)) {

for(int j = 0; j < table.get(i).size(); j++) {

if(!table.get(i).get(j).type.equals("NUM") && id.containsKey((table.get(i).get(j).name))) {

ArrayList<Symbol> temp = new ArrayList<>();

temp.addAll(table.get(i));

temp.add(table.get(i).get(j));

id.put(table.get(i).get(j).name, temp);

}

else if(!table.get(i).get(j).type.equals("NUM") && !id.containsKey(table.get(i).get(j).name)) {

ArrayList<Symbol> temp = new ArrayList<>();

temp.add(table.get(i).get(j));

id.put(table.get(i).get(j).name, temp);

}

else if(table.get(i).get(j).type.equals("NUM")) {

cons.put(table.get(i).get(j).name, table.get(i).get(j));

}

}

}

}

}

// Utility function to print the table

public void printTable() {

for(int i = 0; i < 6; i++) {

if(table.containsKey(i)) {

for(int j = 0; j < table.get(i).size(); j++) {

System.out.println(table.get(i).get(j).type + "\t" + table.get(i).get(j).level + "\t" + table.get(i).get(j).name + "\t");

}

}

}

}

// Pretty print the table along with details

public void printDetails() {

Iterator<Entry<String, ArrayList<Symbol>>> iter = id.entrySet().iterator();

while(iter.hasNext()) {

Entry<String, ArrayList<Symbol>> pair = iter.next();

//System.out.print(pair.getKey() + " = ");

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

if(pair.getValue().get(i).type.equals("int") || pair.getValue().get(i).type.equals("float"))

System.out.println("Variable : " + pair.getKey() + " [Type : " + pair.getValue().get(i).type + "] Declaration => (X\_POS : " + pair.getValue().get(i).xpos + " YPOS : " + pair.getValue().get(i).ypos + ")");

else

System.out.println("Variable : " + pair.getKey() + " | Usage => (X\_POS : " + pair.getValue().get(i).xpos + " YPOS : " + pair.getValue().get(i).ypos + ")");

}

iter.remove();

}

}

// Test Function

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

SymbolTable st = new SymbolTable("output.txt");

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

System.out.println(st.symbol.get(i).token);

}

st.generate();

st.printTable();

st.printDetails()

}

/\*

\* created By : Gaurav Mitra

\*

\* This Class is used to test on Input programs

\*

\* \*/

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Stack;

public class Test {

// stack : Stack for the LL1 parser

// symbol : Saves the output of the Tokeniser and the Symbol Table

// comp : A Compiler class Object

Stack<String> stack = new Stack<>();

ArrayList<Item> symbol = new ArrayList<>();

Compile comp = new Compile();

Test(String filename) throws IOException {

File file = new File(filename);

FileReader fr = new FileReader(file);

BufferedReader br = new BufferedReader(fr);

while(true) {

String line = br.readLine();

if(line == null)

break;

else {

Item i = new Item(line);

symbol.add(i);

}

}

comp.compute();

br.close();

stack.push("program");

//System.out.print(comp.ptable);

}

void compute() {

int pos = 0;

try {

while(!stack.empty() && pos < symbol.size()) {

String top = stack.peek();

String input = symbol.get(pos).token;

System.out.println(stack.toString() + "\t\t : " + input);

if(input.equals(top)) {

pos++;

stack.pop();

}

else {

stack.pop();

String prod = comp.ptable.get(top).get(input);

String arr[] = prod.split(" ");

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

stack.push(arr[i]);

}

}

}

catch(Exception e) {

System.out.print("Compilation Error");

return;

}

System.out.print("String Accepted");

}

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

Test t = new Test("b.txt");

t.compute();

}

}

/\*\*\* Author @ Gaurav Mitra \*\*\*/

/\*\*\* A lexical analyser for restricted C language \*\*\*/

/\*\*\* Key features :

1. Tokenizes C program

2. Identifies the Token types namely KEYWORDS, IDENTIFIERS,

LITERALS, ASSIGN OP, RELATIONAL OP, ARITHMETIC OP,

WHITE SPACE CHARS, DELIMITERS.

3. PROCESSES COMMENTS AND IGNORES THEM.

4. CREATES A SYMBOL TABLE.

5. DISPLAYS ROW AND COLUMN POSITION OF LEXEME.

\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define BUFFER\_SIZE 10000

typedef enum types {

KEYWORD = 1,

IDENTIFIER = 2,

ARITHMETIC = 3,

RELATIONAL = 4,

ASSIGN = 5,

DELIMITER = 6,

NUM = 7

}type;

struct symbol {

unsigned int id;

int xpos;

int ypos;

char\* lex;

}SYMTAB[1000];

int PC = 0;

char\* keywords[] = {"else","if","for","int","float","void","return","scan","print","main"};

char arith\_op[] = {'+' , '-' , '\*' , '/'};

char assign\_op = '=';

char relat\_op[] = {'<','>'};

char ws[] = {' ','\t','\n'};

char delim[] = {'(', ')', ';', ',','{','}'};

char \*buffer;

int breaks[100];

int lines = 1;

void readFile(char \*filename) {

FILE \*fp;

breaks[0] = 0;

char line[200];

fp = fopen(filename,"r");

//int count = 0;

while(fgets(line,255,(FILE\*)fp)) {

//printf("%d\n",strlen(line));

int i;

for(i = 0; i < sizeof(line); i++){

if(line[i] == '/' && line[i + 1] == '/') {

break;

}

}

line[i] = '\0';

//strncpy(dest,line,slash);

//strncpy(line,dest,sizeof(dest));

strcat(buffer,line);

breaks[lines + 1] = breaks[lines] + strlen(line);

lines++;

}

}

int is\_arithop(char c) {

int i;

for(i = 0; i < 3; i++)

if(arith\_op[i] == c)

return 1;

return 0;

}

int is\_assignop(char c) {

if(assign\_op == c)

return 1;

else

return 0;

}

int is\_relatop(char c) {

int i;

for(i = 0; i < 2; i++)

if(relat\_op[i] == c)

return 1;

return 0;

}

int is\_delim(char c) {

int i;

for(i = 0; i < 5; i++)

if(delim[i] == c)

return 1;

return 0;

}

int is\_ws(char c) {

int i;

for(i = 0; i < 2; i++)

if(ws[i] == c)

return 1;

return 0;

}

int is\_keyword(char \*token) {

int i;

for(i = 0; i < 10; i++) {

if(!strcmp(token,keywords[i])) {

return 1;

}

}

return 0;

}

int calculateYpos(int x) {

//printf("%d\n",x);

int i;

for(i = 2; i < lines; i++) {

if(breaks[i] > x) {

//printf("%d : %d\n",breaks[i],x);

return i - 1;

}

}

}

int checkNUM(char \*token) {

int i;

int len = strlen(token);

//printf("%s : %d\n",token,len);

if(isdigit(token[0]))

return 1;

if(token[0] == '\"')

return 1;

return 0;

}

void addNode(type var,char \*token,int x) {

int result = 0;

if(strlen(token) == 0 || token[0] == 10)

return;

if(var == IDENTIFIER) {

result = checkNUM(token);

//printf("result : %d",result);

}

SYMTAB[PC].lex = (char\*)malloc(sizeof(char)\*100);

if (!result)

SYMTAB[PC].id = var;

else

SYMTAB[PC].id = NUM;

SYMTAB[PC].ypos = calculateYpos(x);

SYMTAB[PC].xpos = x - breaks[SYMTAB[PC].ypos];

strcpy(SYMTAB[PC].lex,token);

PC++;

}

void addNodeChar(type var,char token,int x) {

char \*p = malloc(sizeof(char)\*3);

\*p = token;

\*(p + 1) = '\0';

SYMTAB[PC].lex = (char\*)malloc(sizeof(char)\*10);

SYMTAB[PC].id = var;

//SYMTAB[PC].xpos = x;

SYMTAB[PC].ypos = calculateYpos(x);

SYMTAB[PC].xpos = x - breaks[SYMTAB[PC].ypos];

strcpy(SYMTAB[PC].lex,p);

PC++;

}

void printSymbolTable() {

FILE \*f;

f = fopen("b.txt","w");

/\*fprintf(f,"\t\tSYMBOL TABLE\n");

fprintf(f,"\t\t------ -----\n\n");

fprintf(f,"ID\tTOKEN\t\tLEXEME\tXPOS\tYPOS\n");

fprintf(f,"--\t-----\t\t------\t----\t----\n");\*/

int i;

for(i = 0; i < PC; i++) {

char typ[15];

switch(SYMTAB[i].id) {

case 1 :

strcpy(typ,"KEYWORD");

break;

case 2 :

strcpy(typ,"IDENTIFIER");

break;

case 3 :

strcpy(typ,"ARITHMETIC");

break;

case 4 :

strcpy(typ,"RELATIONAL");

break;

case 5 :

strcpy(typ,"ASSIGN");

break;

case 6 :

strcpy(typ,"DELIMITER");

break;

case 7 :

strcpy(typ,"NUM");

}

if(SYMTAB[i].id == KEYWORD || SYMTAB[i].id == NUM)

fprintf(f,"%d\t%s\t%s\t%d\t%d\n",SYMTAB[i].id,typ,SYMTAB[i].lex,SYMTAB[i].xpos,SYMTAB[i].ypos);

else

fprintf(f,"%d\t%s\t%s\t%d\t%d\n",SYMTAB[i].id,typ,SYMTAB[i].lex,SYMTAB[i].xpos,SYMTAB[i].ypos);

/\*if(SYMTAB[i].id == 2) {

printf("%c %c\n",SYMTAB[i].lex[0],SYMTAB[i].lex[1]);

}\*/

}

fclose(f);

}

int main(void) {

int global\_xpos = 0;

char \*filename = (char\*)malloc(sizeof(char)\*100);

printf("Enter the filename : ",filename);

scanf("%s",filename);

buffer = (char\*)malloc(sizeof(char)\*BUFFER\_SIZE);

readFile(filename);

int len = strlen(buffer);

int tok\_pos = 0, i;

char \*token = (char\*)malloc(150);

token[0] = '\0';

for(i = 3; i < len; i++) {

global\_xpos++;

char p = buffer[i];

if(is\_arithop(p) || is\_assignop(p) || is\_relatop(p) || is\_delim(p)) {

if(tok\_pos > 0 && is\_keyword(token)) {

// printf("Keyword : %s\n",token);

addNode(KEYWORD,token,global\_xpos);

token[0] = '\0';

tok\_pos = 0;

}

else if(tok\_pos > 0) {

// printf("Identifier : %s\n",token);

addNode(IDENTIFIER,token,global\_xpos);

token[0] = '\0';

tok\_pos = 0;

}

if(is\_arithop(p)) {

addNodeChar(ARITHMETIC,p,global\_xpos);

// printf("Arith op : %c\n",p);

}

else if(is\_assignop(p)) {

addNodeChar(ASSIGN,p,global\_xpos);

// printf("Assign op : %c\n",p);

}

else if(is\_relatop(p)) {

addNodeChar(RELATIONAL,p,global\_xpos);

// printf("Relational op : %c\n",p);

}

else if(is\_delim(p)) {

addNodeChar(DELIMITER,p,global\_xpos);

// printf("Delimiter op : %c\n",p);

}

}

else {

if(is\_ws(p)) {

if(is\_keyword(token)) {

// printf("Keyword : %s\n",token);

addNode(KEYWORD,token,global\_xpos);

token[0] = '\0';

tok\_pos = 0;

}

else {

// printf("Identifier : %s\n",token);

addNode(IDENTIFIER,token,global\_xpos);

token[0] = '\0';

tok\_pos = 0;

}

}

else {

token[tok\_pos++] = p;

token[tok\_pos] = '\0';

}

}

}

printSymbolTable();

}

Grammar

program -> declaration program | main\_func

main\_func -> ret\_type main ( cmd\_args ) { exe }

cmd\_args -> void | e$

ret\_type -> int | void

declaration -> type id\_list ;

type -> int | float

id\_list -> id\_decl | id\_decl , id\_list

id\_decl -> id | id = num

exe -> stmt exe | e$

stmt -> for\_stmt | if\_stmt | read\_stmt | write\_stmt | int id\_list | float id\_list | assignment | expr ;

assignment -> id assignop expr ;

assignop -> =

write\_stmt -> print ( expr ) ;

read\_stmt -> scan ( id ) ;

for\_stmt -> for ( optional\_expr ; optional\_expr ; optional\_expr ) sub\_block

if\_stmt -> if ( expr ) sub\_block

sub\_block -> { exe } | for ( optional\_expr ; optional\_expr ; optional\_expr ) sub\_block | if ( expr ) sub\_block | scan ( id ) ; | print ( expr ) ; | type id\_list | id assignop expr ; | expr ;

optional\_expr -> expr | e$

expr -> id assignop expr ; | R relop expr | R

R -> term addop R | term

term -> factor multop term | factor

factor -> ( expr ) | id | num

relop -> > | < | >= | <= | == | !=

addop -> + | -

multop -> \* | / | %

First of grammar

optional\_expr -> [id e$ num (]

relop -> [== <= != >= > = < e$]

expr -> [id num (]

type -> [int float]

sub\_block -> [id num for int if ( scan { float print]

read\_stmt -> [scan]

addop -> [+ -]

R -> [id num (]

cmd\_args -> [e$ void]

multop -> [% \* /]

assignop -> [=]

for\_stmt -> [for]

declaration -> [int float]

stmt -> [id num for int if ( scan float print]

id\_decl -> [id]

ret\_type -> [int void]

factor -> [id num (]

id\_list -> [id]

write\_stmt -> [print]

assignment -> [id]

term -> [id num (]

main\_func -> [int void]

program -> [int float]

if\_stmt -> [if]

exe -> [id e$ num for if int scan ( float print]

Follow of grammar

FOLLOW OF GRAMMAR

-----------------

optional\_expr -> [; )]

relop -> [; ) id num]

expr -> [; )]

type -> [id]

sub\_block -> [for if read print int float id ( num }]

read\_stmt -> [for if read print int float id ( num }]

addop -> [id, num, (]

R -> [== <= != >= > = < ; )]

cmd\_args -> [)]

multop -> [id, num, (]

assignop -> [id num (]

for\_stmt -> [for if read print int float id ( num }]

declaration -> [int, float,void]

stmt -> [for if read print int float id ( num }]

id\_decl -> [, ; for if read print int float id ( num }]

ret\_type -> [main]

factor -> [== <= != >= > = < ; ) + - % \* /]

id\_list -> [; for if read print int float id ( num }]

write\_stmt -> [for if read print int float id ( num }]

assignment -> [follow(expr)]

term -> [== <= != >= > = < ; ) + -]

main\_func -> [$]

program -> [$]

if\_stmt -> [for if read print int float id ( num }]

exe -> [}]

Symbol table

Variable : a [Type : float] Declaration => (X\_POS : 7 YPOS : 3) Scope => 1

Variable : a | Usage => (X\_POS : 10 YPOS : 5)

Variable : a | Usage => (X\_POS : 10 YPOS : 7)

Variable : a | Usage => (X\_POS : 10 YPOS : 5)

Variable : b [Type : float] Declaration => (X\_POS : 10 YPOS : 3) Scope => 1

Variable : b | Usage => (X\_POS : 4 YPOS : 8)

Variable : b | Usage => (X\_POS : 9 YPOS : 10)

Variable : b | Usage => (X\_POS : 10 YPOS : 11)

Variable : b | Usage => (X\_POS : 6 YPOS : 8)

Variable : c [Type : float] Declaration => (X\_POS : 10 YPOS : 7) Scope => 2

Variable : c | Usage => (X\_POS : 8 YPOS : 8)

Variable : c | Usage => (X\_POS : 9 YPOS : 10)

Variable : c | Usage => (X\_POS : 10 YPOS : 11)

PARSING table

[ optional\_expr , id ] => [ optional\_expr -> expr ]

[ optional\_expr , e$ ] => [ optional\_expr -> e$ ]

[ optional\_expr , num ] => [ optional\_expr -> expr ]

[ optional\_expr , (] ] => [ optional\_expr -> expr ]

[ optional\_expr , ;] ] => [ optional\_expr -> ; ]

[ relop , == ] => [ relop -> == ]

[ relop , <= ] => [ relop -> <= ]

[ relop , != ] => [ relop -> != ]

[ relop , >= ] => [ relop -> >= ]

[ relop , > ] => [ relop -> > ]

[ relop , = ] => [ relop -> = ]

[ relop , < ] => [ relop -> < ]

[ relop , e$ ] => [ relop -> e$ ]

[ expr , id ] => [ expr -> R ]

[ expr , num ] => [ expr -> R ]

[ expr , (] ] => [ expr -> R ]

[ type , int ] => [ type -> int ]

[ type , float ] => [ type -> float ]

[ read\_stmt , scan ] => [ read\_stmt -> scan ( id ) ; ]

[ sub\_block , id ] => [ sub\_block -> expr ; ]

[ sub\_block , num ] => [ sub\_block -> expr ; ]

[ sub\_block , for ] => [ sub\_block -> for ( optional\_expr ; optional\_expr ; optional\_expr ) sub\_block ]

[ sub\_block , int ] => [ sub\_block -> type id\_list ]

[ sub\_block , if ] => [ sub\_block -> if ( expr ) sub\_block ]

[ sub\_block , scan ] => [ sub\_block -> scan ( id ) ; ]

[ sub\_block , float] ] => [ sub\_block -> type id\_list ]

[ sub\_block , (] ] => [ sub\_block -> expr ; ]

[ sub\_block , { ] => [ sub\_block -> { exe } ]

[ sub\_block , print ] => [ sub\_block -> print ( expr ) ; ]

[ addop , + ] => [ addop -> + ]

[ addop , - ] => [ addop -> - ]

[ R , id ] => [ R -> term ]

[ R , num ] => [ R -> term ]

[ R , (] ] => [ R -> term ]

[ cmd\_args , e$ ] => [ cmd\_args -> e$ ]

[ cmd\_args , void ] => [ cmd\_args -> void ]

[ assignop , = ] => [ assignop -> = ]

[ multop , % ] => [ multop -> % ]

[ multop , \* ] => [ multop -> \* ]

[ multop , / ] => [ multop -> / ]

[ for\_stmt , for ] => [ for\_stmt -> for ( optional\_expr ; optional\_expr ; optional\_expr ) sub\_block ]

[ declaration , int ] => [ declaration -> type id\_list ; ]

[ declaration , float] ] => [ declaration -> type id\_list ; ]

[ stmt , id ] => [ stmt -> expr ; ]

[ stmt , num ] => [ stmt -> expr ; ]

[ stmt , for ] => [ stmt -> for\_stmt ]

[ stmt , int ] => [ stmt -> int id\_list ]

[ stmt , scan] ] => [ stmt -> read\_stmt ]

[ stmt , if ] => [ stmt -> if\_stmt ]

[ stmt , print] ] => [ stmt -> write\_stmt ]

[ stmt , (] ] => [ stmt -> expr ; ]

[ stmt , id] ] => [ stmt -> assignment ]

[ stmt , float ] => [ stmt -> float id\_list ]

[ id\_decl , id ] => [ id\_decl -> id = num ]

[ ret\_type , int ] => [ ret\_type -> int ]

[ ret\_type , void ] => [ ret\_type -> void ]

[ factor , id ] => [ factor -> id ]

[ factor , num ] => [ factor -> num ]

[ factor , ( ] => [ factor -> ( expr ) ]

[ id\_list , id ] => [ id\_list -> id\_decl , id\_list ]

[ write\_stmt , print ] => [ write\_stmt -> print ( expr ) ; ]

[ assignment , id ] => [ assignment -> id assignop expr ; ]

[ term , id ] => [ term -> factor relop factor ]

[ term , num ] => [ term -> factor ]

[ term , (] ] => [ term -> factor ]

[ main\_func , int ] => [ main\_func -> ret\_type main ( cmd\_args ) { exe } ]

[ main\_func , void] ] => [ main\_func -> ret\_type main ( cmd\_args ) { exe } ]

[ program , int ] => [ program -> main\_func ]

[ program , void] ] => [ program -> main\_func ]

[ program , float] ] => [ program -> declaration program ]

[ if\_stmt , if ] => [ if\_stmt -> if ( expr ) sub\_block ]

[ exe , id ] => [ exe -> stmt exe ]

[ exe , e$ ] => [ exe -> e$ ]

[ exe , num ] => [ exe -> stmt exe ]

[ exe , for ] => [ exe -> stmt exe ]

[ exe , int ] => [ exe -> stmt exe ]

[ exe , if ] => [ exe -> stmt exe ]

[ exe , print] ] => [ exe -> stmt exe ]

[ exe , ( ] => [ exe -> stmt exe ]

[ exe , scan ] => [ exe -> stmt exe ]

[ exe , float ] => [ exe -> stmt exe ]

Input code

int main() {

int id;

scan(id);

if(id < id) {

print(id);

}

for(id=id;id>id;) {

id=id-id;

}

}

PARSER OUTPUT

$ program int [ program , int ] => [ program -> main\_func ]

$ main\_func int [ main\_func , int ] => [ main\_func -> ret\_type main ( cmd\_args ) { exe } ]

$ { exe } ) cmd\_args ( main ret\_type int [ ret\_type , int ] => [ ret\_type -> int ]

$ { exe } ) cmd\_args ( main int int Match

$ { exe } ) cmd\_args ( main main Match

$ } exe { ) cmd\_args ( ( Match

$ } exe { ) cmd\_args void [ cmd\_args , void ] => [ cmd\_args -> void ]

$ } exe { ) void void Match

$ } exe { ) ) Match

$ } exe { { Match

$ } exe int [ exe , int ] => [ exe -> stmt exe ]

$ } exe stmt id [ stmt , id ] => [ stmt -> expr ; ]

$ } exe ; expr id [ expr , id ] => [ expr -> R ]

$ } exe ; R id [ R , id ] => [ R -> term ]

$ } exe ; term id [ term , id ] => [ term -> factor ]

$ } exe ; factor id [ factor , id ] => [ factor -> id ]

$ } exe ; id id Match

$ } exe ; ; Match

$ } exe scan [ exe , scan ] => [ exe -> stmt exe ]

$ } exe stmt scan [ stmt , scan] ] => [ stmt -> read\_stmt ]

$ } exe read\_stmt scan [ read\_stmt , scan ] => [ read\_stmt -> scan ( id ) ; ]

$ } exe ; ) id ( scan scan Match

$ } exe ; ) id ( ( Match

$ } exe ; ) id id Match

$ } exe ; ) ) Match

$ } exe ; ; Match

$ } exe if [ exe , if ] => [ exe -> stmt exe ]

$ } exe stmt if [ stmt , if ] => [ stmt -> if\_stmt ]

$ } exe if\_stmt if [ if\_stmt , if ] => [ if\_stmt -> if ( expr ) sub\_block ]

$ } exe sub\_block ) expr ( if if Match

$ } exe sub\_block ) expr ( ( Match

$ } exe sub\_block ) expr id [ expr , id ] => [ expr -> R ]

$ } exe sub\_block ) R id [ R , id ] => [ R -> term ]

$ } exe sub\_block ) term id [ term , id ] => [ term -> factor relop factor]

$ } exe sub\_block ) factor relop factor id [ factor , id ] => [ factor -> id ]

$ } exe sub\_block ) factor relop id id Match

$ } exe sub\_block ) factor relop < [ relop , < ] => [ relop -> < ]

$ } exe sub\_block ) factor < < Match

$ } exe sub\_block ) factor id [ factor , id ] => [ factor -> id ]

$ } exe sub\_block ) id id Match

$ } exe sub\_block ) ) Match

$ } exe sub\_block { [ sub\_block , { ] => [ sub\_block -> { exe } ]

$ } exe } exe { { Match

$ } exe } exe print [ exe , print] ] => [ exe -> stmt exe ]

$ } exe } exe stmt print [ stmt , print] ] => [ stmt -> write\_stmt ]

$ } exe } exe write\_stmt print [ write\_stmt , print ] => [ write\_stmt -> print ( expr ) ; ]

$ } exe } exe ; ) expr ( print print Match

$ } exe } exe ; ) expr ( ( Match

$ } exe } exe ; ) expr id [ expr , id ] => [ expr -> R ]

$ } exe } exe ; ) R id [ R , id ] => [ R -> term ]

$ } exe } exe ; ) term id [ term , id ] => [ term -> factor]

$ } exe } exe ; ) factor id [ factor , id ] => [ factor -> id ]

$ } exe } exe ; ) id id Match

$ } exe } exe ; ) ) Match

$ } exe } exe ; ; Match

$ } exe } exe } [ exe , e$ ] => [ exe -> e$ ]

$ } exe } } Match

$ } exe for [ exe , for ] => [ exe -> stmt exe ]

$ } exe stmt for [ stmt , for ] => [ stmt -> for\_stmt ]

$ } exe for\_stmt for [ for\_stmt , for ] => [ for\_stmt -> for ( optional\_expr ; optional\_expr ; optional\_expr ) sub\_block ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; optional\_expr ( for for Match

$ } exe sub\_block ) optional\_expr ; optional\_expr ; optional\_expr ( ( Match

$ } exe sub\_block ) optional\_expr ; optional\_expr ; optional\_expr id [ optional\_expr , id ] => [ optional\_expr -> expr ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; expr id [ expr , id ] => [ expr -> R ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; R id [ R , id ] => [ R -> term ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; term id [ term , id ] => [ term -> factor relop factor ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; factor relop factor id [ factor , id ] => [ factor -> id ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; factor relop id id Match

$ } exe sub\_block ) optional\_expr ; optional\_expr ; factor relop = [ relop , = ] => [ relop -> = ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; factor = = Match

$ } exe sub\_block ) optional\_expr ; optional\_expr ; factor id [ factor , id ] => [ factor -> id ]

$ } exe sub\_block ) optional\_expr ; optional\_expr ; id id Match

$ } exe sub\_block ) optional\_expr ; optional\_expr ; ; Match

$ } exe sub\_block ) optional\_expr ; optional\_expr id [ optional\_expr , id ] => [ optional\_expr -> expr ]

$ } exe sub\_block ) optional\_expr ; expr id [ expr , id ] => [ expr -> R ]

$ } exe sub\_block ) optional\_expr ; R id [ R , id ] => [ R -> term ]

$ } exe sub\_block ) optional\_expr ; term id [ term , id ] => [ term -> factor relop factor ]

$ } exe sub\_block ) optional\_expr ; factor relop factor id [ factor , id ] => [ factor -> id ]

$ } exe sub\_block ) optional\_expr ; factor relop id id Match

$ } exe sub\_block ) optional\_expr ; factor relop < [ relop , < ] => [ relop -> < ]

$ } exe sub\_block ) optional\_expr ; factor < < Match

$ } exe sub\_block ) optional\_expr ; factor id [ factor , id ] => [ factor -> id ]

$ } exe sub\_block ) optional\_expr ; id id Match

$ } exe sub\_block ) optional\_expr ; ; Match

$ } exe sub\_block ) optional\_expr ; [ optional\_expr , ;] ] => [ optional\_expr -> ; ]

$ } exe sub\_block ) ; ; Match

$ } exe sub\_block ) ) Match

$ } exe sub\_block id [ sub\_block , id ] => [ sub\_block -> expr ; ]

$ } exe ; expr id [ expr , id ] => [ expr -> R ]

$ } exe ; R id [ R , id ] => [ R -> term ]

$ } exe ; term id [ term , id ] => [ term -> factor relop factor ]

$ } exe ; factor relop factor id [ factor , id ] => [ factor -> id ]

$ } exe ; factor relop id id Match

$ } exe ; factor relop = [ relop , = ] => [ relop -> = ]

$ } exe ; factor = = Match

$ } exe ; factor id [ factor , id ] => [ factor -> id ]

$ } exe ; ; Match

$ } exe } [ exe , e$ ] => [ exe -> e$ ]

$ } } Match

String Accepted.