**Assignment 1**

**Aim :**

Generate Symbol table, Literal table, Pool table & Intermediate code of a two-pass Assembler for the given source code. also, display errors.

**Code:**

TableRow.java

public class TableRow {

String symbol;

int addess,index;

public String getSymbol() {

return symbol;

}

public TableRow(String symbol, int addess) {

super();

this.symbol = symbol;

this.addess = addess;

index=0;

}

public void setSymbol(String symbol) {

this.symbol = symbol;

}

public TableRow(String symbol, int addess, int index) {

super();

this.symbol = symbol;

this.addess = addess;

this.index = index;

}

public int getAddess() {

return addess;

}

public void setAddess(int addess) {

this.addess = addess;

}

public int getIndex() {

return index;

}

public void setIndex(int index) {

this.index = index;

}

}

INSTtable.java

import java.util.HashMap;

public class INSTtable {

HashMap<String, Integer> AD,RG,IS,CC,DL;

public INSTtable()

{

AD=new HashMap<>();

CC = new HashMap<>();

IS = new HashMap<>();

RG = new HashMap<>();

DL=new HashMap<String, Integer>();

DL.put("DC", 01);

DL.put("DS", 02);

IS.put("STOP",0);

IS.put("ADD",1);

IS.put("SUB",2);

IS.put("MULT",3);

IS.put("MOVER",4);

IS.put("MOVEM",5);

IS.put("COMP",6);

IS.put("BC",7);

IS.put("DIV",8);

IS.put("READ",9);

IS.put("PRINT",10);

CC.put("LT",1);

CC.put("LE",2);

CC.put("EQ",3);

CC.put("GT",4);

CC.put("GE",5);

CC.put("ANY",6);

AD.put("START",1);

AD.put("END",2);

AD.put("ORIGIN",3);

AD.put("EQU",4);

AD.put("LTORG",5);

RG.put("AREG",1);

RG.put("BREG",2);

RG.put("CREG",3);

RG.put("DREG",4);

}

public String getType(String s)

{

s=s.toUpperCase();

if(AD.containsKey(s))

return "AD";

else if(IS.containsKey(s))

return "IS";

else if(CC.containsKey(s))

return "CC";

else if(DL.containsKey(s))

return "DL";

else if(RG.containsKey(s))

return "RG";

return "";

}

public int getCode(String s)

{

s = s.toUpperCase();

if(AD.containsKey(s))

return AD.get(s);

else if(IS.containsKey(s))

return IS.get(s);

else if(CC.containsKey(s))

return CC.get(s);

else if(DL.containsKey(s))

return DL.get(s);

else if(RG.containsKey(s))

return RG.get(s);

return -1;

}

}

PassOne.java

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.LinkedHashMap;

public class PassOne {

int lc=0;

int libtab\_ptr=0,pooltab\_ptr=0;

int symIndex=0,litIndex=0;

LinkedHashMap<String, TableRow> SYMTAB;

ArrayList<TableRow> LITTAB;

ArrayList<Integer> POOLTAB;

private BufferedReader br;

public PassOne()

{

SYMTAB =new LinkedHashMap<>();

LITTAB=new ArrayList<>();

POOLTAB=new ArrayList<>();

lc=0;

POOLTAB.add(0);

}

public static void main(String[] args) {

PassOne one=new PassOne();

try

{

one.parseFile();

}

catch (Exception e) {

System.out.println("Error: "+e);// TODO: handle exception

}

}

public void parseFile() throws Exception

{String prev="";

String line,code;

br = new BufferedReader(new FileReader("input.asm"));

BufferedWriter bw=new BufferedWriter(new FileWriter("IC.txt"));

INSTtable lookup=new INSTtable();

while((line=br.readLine())!=null)

{

String parts[]=line.split("\\s+");

if(!parts[0].isEmpty()) //processing of label

{

if(SYMTAB.containsKey(parts[0]))

SYMTAB.put(parts[0], new TableRow(parts[0], lc, SYMTAB.get(parts[0]).getIndex()));

else

SYMTAB.put(parts[0],new TableRow(parts[0], lc, ++symIndex));

}

if(parts[1].equals("LTORG"))

{

int ptr=POOLTAB.get(pooltab\_ptr);

for(int j=ptr;j<libtab\_ptr;j++)

{

lc++;

LITTAB.set(j, new TableRow(LITTAB.get(j).getSymbol(),lc));

code="(DL,01)\t(C,"+LITTAB.get(j).symbol+")";

bw.write(code+"\n");

}

pooltab\_ptr++;

POOLTAB.add(libtab\_ptr);

}

if(parts[1].equals("START"))

{

lc=expr(parts[2]);

code="(AD,01)\t(C,"+lc+")";

bw.write(code+"\n");

prev="START";

}

else if(parts[1].equals("ORIGIN"))

{

lc=expr(parts[2]);

String splits[]=parts[2].split("\\+"); //Same for - SYMBOL //Add code

code="(AD,03)\t(S,"+SYMTAB.get(splits[0]).getIndex()+")+"+Integer.parseInt(splits[1]);

bw.write(code+"\n");

}

//Now for EQU

if(parts[1].equals("EQU"))

{

int loc=expr(parts[2]);

//below If conditions are optional as no IC is generated for them

if(parts[2].contains("+"))

{

String splits[]=parts[2].split("\\+");

code="(AD,04)\t(S,"+SYMTAB.get(splits[0]).getIndex()+")+"+Integer.parseInt(splits[1]);

}

else if(parts[2].contains("-"))

{

String splits[]=parts[2].split("\\-");

code="(AD,04)\t(S,"+SYMTAB.get(splits[0]).getIndex()+")-"+Integer.parseInt(splits[1]);

}

else

{

code="(AD,04)\t(C,"+Integer.parseInt(parts[2]+")");

}

bw.write(code+"\n");

if(SYMTAB.containsKey(parts[0]))

SYMTAB.put(parts[0], new TableRow(parts[0],loc,SYMTAB.get(parts[0]).getIndex())) ;

else

SYMTAB.put(parts[0], new TableRow(parts[0],loc,++symIndex));

}

if(parts[1].equals("DC"))

{

lc++;

int constant=Integer.parseInt(parts[2].replace("'",""));

code="(DL,01)\t(C,"+constant+")";

bw.write(code+"\n");

}

else if(parts[1].equals("DS"))

{

int size=Integer.parseInt(parts[2].replace("'", ""));

code="(DL,02)\t(C,"+size+")";

bw.write(code+"\n");

/\*if(prev.equals("START"))

{

lc=lc+size-1;//System.out.println("here");

}

else

\*/ lc=lc+size;

prev="";

}

if(lookup.getType(parts[1]).equals("IS"))

{

code="(IS,0"+lookup.getCode(parts[1])+")\t";

int j=2;

String code2="";

while(j<parts.length)

{

parts[j]=parts[j].replace(",", "");

if(lookup.getType(parts[j]).equals("RG"))

{

code2+=lookup.getCode(parts[j])+"\t";

}

else

{

if(parts[j].contains("="))

{

parts[j]=parts[j].replace("=", "").replace("'", "");

LITTAB.add(new TableRow(parts[j], -1,++litIndex));

libtab\_ptr++;

code2+="(L,"+(litIndex)+")";

}

else if(SYMTAB.containsKey(parts[j]))

{

int ind=SYMTAB.get(parts[j]).getIndex();

code2+= "(S,0"+ind+")";

}

else

{

SYMTAB.put(parts[j], new TableRow(parts[j],-1,++symIndex));

int ind=SYMTAB.get(parts[j]).getIndex();

code2+= "(S,0"+ind+")";

}

}

j++;

}

lc++;

code=code+code2;

bw.write(code+"\n");

}

if(parts[1].equals("END"))

{

int ptr=POOLTAB.get(pooltab\_ptr);

for(int j=ptr;j<libtab\_ptr;j++)

{

lc++;

LITTAB.set(j, new TableRow(LITTAB.get(j).getSymbol(),lc));

code="(DL,01)\t(C,"+LITTAB.get(j).symbol+")";

bw.write(code+"\n");

}

pooltab\_ptr++;

POOLTAB.add(libtab\_ptr);

code="(AD,02)";

bw.write(code+"\n");

}

}

bw.close();

printSYMTAB();

//Printing Literal table

PrintLITTAB();

printPOOLTAB();

}

void PrintLITTAB() throws IOException

{

BufferedWriter bw=new BufferedWriter(new FileWriter("LITTAB.txt"));

System.out.println("\nLiteral Table\n");

//Processing LITTAB

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

{

TableRow row=LITTAB.get(i);

System.out.println(i+"\t"+row.getSymbol()+"\t"+row.getAddess());

bw.write((i+1)+"\t"+row.getSymbol()+"\t"+row.getAddess()+"\n");

}

bw.close();

}

void printPOOLTAB() throws IOException

{

BufferedWriter bw=new BufferedWriter(new FileWriter("POOLTAB.txt"));

System.out.println("\nPOOLTAB");

System.out.println("Index\t#first");

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

System.out.println(i+"\t"+POOLTAB.get(i));

bw.write((i+1)+"\t"+POOLTAB.get(i)+"\n");

}

bw.close();

}

void printSYMTAB() throws IOException

{

BufferedWriter bw=new BufferedWriter(new FileWriter("SYMTAB.txt"));

//Printing Symbol Table

java.util.Iterator<String> iterator = SYMTAB.keySet().iterator();

System.out.println("SYMBOL TABLE");

while (iterator.hasNext()) {

String key = iterator.next().toString();

TableRow value = SYMTAB.get(key);

System.out.println(value.getIndex()+"\t" + value.getSymbol()+"\t"+value.getAddess());

bw.write(value.getIndex()+"\t" + value.getSymbol()+"\t"+value.getAddess()+"\n");

}

bw.close();

}

public int expr(String str)

{

int temp=0;

if(str.contains("+"))

{

String splits[]=str.split("\\+");

temp=SYMTAB.get(splits[0]).getAddess()+Integer.parseInt(splits[1]);

}

else if(str.contains("-"))

{

String splits[]=str.split("\\-");

temp=SYMTAB.get(splits[0]).getAddess()-(Integer.parseInt(splits[1]));

}

else

{

temp=Integer.parseInt(str);

}

return temp;

}

}

**Input:**

**Output:**

**Conclusion:**

Pass 1 of 2 Pass Assembler was implemented by using all the required data structures like symbol table, literal table, pool table etc.