**COMPILER LAB REPORT**

**Assignment V Project 9**

**Class:UG-III Section: A1**

Shaswata Saha (10)

Radib Kar (11)

Anuran Chakraborty (20)

Souvik Saha (27)

1. **Question**

Consider a simple PASCAL-like language with the following structure:

**program {name of the program}**

**uses {comma delimited names of libraries you use}**

**const {global constant declaration block}**

**var {global variable declaration block}**

**function {function declarations, if any}**

**{local variables}**

**begin**

**...**

**end**

**begin {main program block starts}**

**...**

**end. {end of main program block}**

Type declaration can be done as:

**type-identifier-1, type-identifier-2 = type-specifier**

Data Types: integer and real

Input , output statements are in the form **get x** and **put x**

Conditional statement of the form **expression?expression:expression** is supported

Relational operators are supported **{>,<,>=,<=}**

Arithmetic operators supported are **{+,-,\*}**

Part I – Construct a CFG for this language.

Part II – Write lexical analyzer to scan the stream of characters from a program written in the above language and generate stream of tokens.

Part III – Write a top-down parser for this language.

1. **Context Free Grammar**

start -> program id rest1

rest1 -> uses liblist rest2 | rest2

liblist -> id liblist'

liblist' -> , id liblist' | ε

rest2 -> const const\_list rest3

const\_list -> id=num const\_list'

const\_list' -> , id=num const\_list' | ε

rest3 -> var varlist rest4

varlist -> liblist=type;varlist'

varlist' -> liblist=type;liblist' | ε

type -> integer | real

rest4 -> function id varlist rest\_function rest4 | rest\_main

rest\_function -> begin statements end ;

rest\_main -> begin statements end .

statements -> get id; statements | put id ; statements| id=something ; statements | ε

something -> term exp' s3

s3 -> ε | ? exp : exp

exp -> term exp'

exp' -> op term exp' | ε

term -> ID | num

op -> + | - | \* | < | > | z | y

ID -> id

U -> ;

1. **Code**

#include<bits/stdc++.h>

#include <unistd.h>

**using** **namespace** std;

map<char, set<char> > firstSet;

map<char, set<char> > followSet;

map< pair<char,char>, string > table;

int noofProd;

string\* production;

vector<string> prod;

set<char> t,nt;

int conflict;

map<char,string> symbols;

*// Function to populate the symbol mapping*

void populateSym()

{

fstream file2;

string str;

file2.open("mapping.txt", ios::in);

**while**(getline(file2,str))

{

*// string first="";*

*// first+=str[0];*

symbols.insert(make\_pair(str[0],str.substr(2)));

}

file2.close();

}

*// Function to print a production*

void printProd(string prod)

{

int i;

*// cout<<prod<<"\t\t\t\t";*

**if**(prod=="pop" || prod=="scan")

{

cout<<prod;

**return**;

}

string actual="";

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

{

string pr="";

pr+=prod[i];

**if**(i==1)

actual+=" -> ";

**else** **if**(symbols.find(prod[i])==symbols.end())*// Trivial characters*

actual+=pr+" ";

**else**

actual+=symbols[prod[i]]+" ";

}

cout<<actual;

}

*// Function to remove left recursion*

void removeLeftRecur()

{

int i,j;

**for**(i=0;i<noofProd;i++)

{

int nextind=i;

*// If this produciton has a left recursion*

**if**(production[i][0]==production[i][2])

{

*// Then try removing it*

string newprod="";

newprod+=production[i][0];

newprod+='\'';

*// For every produciton having X in the 2 index*

**for**(j=0;j<noofProd;j++)

{

**if**(production[j][0]==production[i][0] && production[j][0]!=production[j][2])

{

*// Remove the first part*

prod.push\_back(production[j]+newprod);

nextind=j;

}

**else** **if**(production[j][0]==production[i][0] && production[j][0]==production[j][2])

{

string nstr=production[j].substr(3);

string nstr2=newprod+'=';

prod.push\_back(nstr2+nstr+newprod);

nextind=j;

}

}

*// Push epsilon*

string ns=newprod+'='+'#';

prod.push\_back(ns);

}

**else**

prod.push\_back(production[i]);

i=nextind;

}

}

*// Function to calculate first for a symbol*

void first(char c, int rule\_no)

{

int j,k;

*// Case for terminal*

**if**(!isupper(c))

{

firstSet[c].insert(c);

}

*// For all the productions*

**for**(j=rule\_no;j<noofProd;j++)

{

**if**(production[j][0]==c)*// If the production has c on LHS then only calclulate*

{

**if**(production[j][2]=='#') *// If production is epsilon then recur for the next symbol*

{

firstSet[c].insert('#');

}

**else** **if**(!isupper(production[j][2])) *// If start symbol is a terminal the first is the start symbol*

{

firstSet[c].insert(production[j][2]);

}

**else** *// If it is a non-terminal then first calculate its firstset*

{

**for**(k=2;k<production[j].length();k++)

{

*// If it is a terminal simply add the terminal*

**if**(!isupper(production[j][k]))

{

firstSet[c].insert(production[j][k]);

**break**;

}

**else**

{

**if**(production[j][k]!=c)

{

*// If it is a nonterminal calculate its first*

first(production[j][k],0);

*// Add the first set to it*

firstSet[c].insert(firstSet[production[j][k]].begin(),firstSet[production[j][k]].end());

*// If epsilon not in this then break*

**if**(firstSet[production[j][k]].find('#')==firstSet[production[j][k]].end())

**break**;

**else**

*// remove #*

firstSet[c].erase('#');

}

**else**

{

*// Check if present symbol first has epsilon*

first(production[j][k],j+1);

**if**(firstSet[production[j][k]].find('#')==firstSet[production[j][k]].end())

**break**;

}

}

}

*// If last contains # add #*

**if**(k==production[j].length())

firstSet[c].insert('#');

}

}

}

}

*// Function to calculate follow*

void follow(char c)

{

int i,j,k;

*// First add $ to follow set of start symbol*

**if**(production[0][0]==c)

followSet[c].insert('$');

*// For every production*

**for**(i=0;i<noofProd;i++)

{

*// Now traverse every production*

**for**(j=2;j<production[i].length();j++)

{

**if**(production[i][j]==c) *// If c found on RHS*

{

**if**(j!=(production[i].length()-1))*// It is not the ending character*

{

*// Insert the first of next non terminal*

followSet[c].insert(firstSet[production[i][j+1]].begin(),firstSet[production[i][j+1]].end());

**for**(k=j+1;k<production[i].length();)

{

**if**(firstSet[production[i][k]].find('#')==firstSet[production[i][k]].end())*// If epsilon does not exist then break*

**break**;

k++;

**if**(k==production[i].length())

**break**;

followSet[c].insert(firstSet[production[i][k]].begin(),firstSet[production[i][k]].end());

}

*// If even the last symbol has epsilon in its first then compute follow of LHS*

**if**(k==production[i].length())

{

**if**(c!=production[i][0])

{

*// Calculate the follow of the Non-Terminal*

*// in the L.H.S. of the production*

follow(production[i][0]);

*// Insert into set*

followSet[c].insert(followSet[production[i][0]].begin(),followSet[production[i][0]].end());

}

}

}

**else**

*// For ending character add follow of LHS*

**if**(j==(production[i].length()-1) && c!=production[i][0])

{

*// Calculate the follow of the Non-Terminal*

*// in the L.H.S. of the production*

follow(production[i][0]);

*// Insert into set*

followSet[c].insert(followSet[production[i][0]].begin(),followSet[production[i][0]].end());

}

}

}

}

}

void fill\_t\_nt(){

vector<string>::iterator i;

string s;

**for**(i=prod.begin();i!=prod.end();i++){

nt.insert((\*i)[0]);

}

t.insert('#');

t.insert('$');

**for**(i=prod.begin();i!=prod.end();i++){

s=(\*i);

**for**(int j=2;j<s.length();j++){

**if**(nt.find(s[j])==nt.end()){

t.insert(s[j]);

}

}

}

}

void make\_table(){

vector<string>::iterator prodit;

string rule;

conflict=0;

**for**(prodit = prod.begin(); prodit!= prod.end(); prodit++){

rule = \*(prodit);

int i;

**for**(i=2;i<rule.length();i++){

set<char> first = firstSet[rule[i]];

set<char>::iterator it;

int flag = 1; *//check whether current character in rhs of rule has epsilon*

**for**(it = first.begin(); it!=first.end(); it++){

**if**((\*it)=='#'){

flag = 0;

}

**else**{

**if**(table.find(make\_pair(rule[0],(\*it)))!=table.end() && table[make\_pair(rule[0],(\*it))]!=rule){

cout<<"Error 1 at "<<rule[0]<<","<<(\*it)<<","<<rule<<","<<table[make\_pair(rule[0],\*it)]<<endl;

conflict=1;

**return**;

}

table[make\_pair(rule[0],(\*it))] = rule;

}

}

**if**(flag){ *//if epsilon is not present, this rule is not needed any more.*

**break**;

}

}

**if**(i == rule.length()){ *//the entire rhs has epsilon in first. so followSet of lhs is used.*

set<char> fol = followSet[rule[0]];

set<char>::iterator it;

**for**(it = fol.begin(); it!=fol.end(); it++){

**if**(table.find(make\_pair(rule[0],(\*it)))!=table.end() && table[make\_pair(rule[0],(\*it))]!=rule){

cout<<"Error 2 at "<<rule[0]<<","<<(\*it)<<","<<rule<<","<<table[make\_pair(rule[0],\*it)]<<endl;

conflict=1;

**return**;

}

table[make\_pair(rule[0],(\*it))] = rule;

}

}

}

set<char>::iterator itt,itnt;

*// cout<<"Non-terminal\tTerminal\tRule\n";*

*// for(itnt = nt.begin(); itnt!=nt.end();itnt++){ //non terminal loop*

*// for(itt = t.begin(); itt!=t.end();itt++){ //terminal loop*

*// cout<<(\*itnt)<<"\t"<<(\*itt)<<"\t"<<table[make\_pair(\*itnt,\*itt)]<<"\n";*

*// }*

*// cout<<endl;*

*// }*

*//set<char>::iterator itt, itnt;*

**for**(itnt = nt.begin(); itnt!=nt.end();itnt++){ *//non terminal loop*

**for**(itt = t.begin(); itt!=t.end();itt++){ *//terminal loop*

**if**(table.count(make\_pair(\*itnt,\*itt))==0){

*//if(table.find( make\_pair( (\*itnt),(\*itt) ) ) == table.end()){*

**if**((\*itt)=='$' || followSet[(\*itnt)].find((\*itt))!=followSet[(\*itnt)].end()){

table[make\_pair((\*itnt),(\*itt))] = "pop";

}

**else** **if**(firstSet[(\*itnt)].find((\*itt))==firstSet[(\*itnt)].end() && followSet[(\*itnt)].find((\*itt))==followSet[(\*itnt)].end()){

table[make\_pair((\*itnt),(\*itt))] = "scan";

}

}

}

}

}

int main(int argc, char **const** \*argv[])

{

int i,j;

printf("Enter number of productions**\n**");

cin>>noofProd;

printf("Enter the productions individually**\n**");

populateSym();

production=**new** string[noofProd];

**for**(i=0;i<noofProd;i++)

cin>>production[i];

removeLeftRecur();

fill\_t\_nt();

*// for(i=0;i<prod.size();i++)*

*// cout<<prod[i]<<endl;*

*// Insert first of terminals*

**for**(i=0;i<noofProd;i++)

**for**(j=0;j<production[i].length();j++)

**if**(!isupper(production[i][j]))*// Terminal*

firstSet[production[i][j]].insert(production[i][j]);

**for**(i=0;i<noofProd;i++)

first(production[i][0],0);

map<char, set<char> >::iterator it;

set<char>::iterator its;

cout<<"printing terminals"<<endl;

**for**(its=t.begin();its!=t.end();its++)

{

**if**(symbols.find(\*its)==symbols.end())

cout<<\*its<<endl;

**else**

cout<<symbols[\*its]<<endl;

}

cout<<"printing non terminals"<<endl;

**for**(its=nt.begin();its!=nt.end();its++)

{

**if**(symbols.find(\*its)==symbols.end())

cout<<\*its<<endl;

**else**

cout<<symbols[\*its]<<endl;

}

*// Printing first set*

**for**(it=firstSet.begin();it!=firstSet.end();it++)

{

cout<<"first(";

**if**(symbols.find(it->first)==symbols.end())

cout<<it->first<<" ";

**else**

cout<<symbols[it->first];

cout<<") : {";

**for**(its=it->second.begin();its!=it->second.end();its++)

{

**if**(symbols.find(\*its)==symbols.end())

cout<<\*its<<" ";

**else**

cout<<symbols[\*its]<<" ";

}

cout<<"}**\n**";

}

cout<<"====================================**\n**";

**for**(i=0;i<noofProd;i++)

follow(production[i][0]);

*// Printing follow set*

**for**(it=followSet.begin();it!=followSet.end();it++)

{

cout<<"follow(";

**if**(symbols.find(it->first)==symbols.end())

cout<<it->first;

**else**

cout<<symbols[it->first];

cout<<") : {";

it->second.erase('#');

**for**(its=it->second.begin();its!=it->second.end();its++)

{

**if**(symbols.find(\*its)==symbols.end())

cout<<\*its<<" ";

**else**

cout<<symbols[\*its]<<" ";

}

cout<<"}**\n**";

}

cout<<"====================================**\n**";

make\_table();

**if**(conflict==0)

{

cout<<"Table making Done**\n**Printing table**\n\n**";

set<char>::iterator itt,itnt;

cout<<"Non-terminal**\t**Terminal**\t**Rule**\n**";

**for**(itnt = nt.begin(); itnt!=nt.end();itnt++){ *//non terminal loop*

**for**(itt = t.begin(); itt!=t.end();itt++)

{ *//terminal loop*

**if**(symbols.find(\*itnt)==symbols.end())

cout<<\*itnt<<"**\t**";

**else**

cout<<symbols[\*itnt]<<"**\t**";

**if**(symbols.find(\*itt)==symbols.end())

cout<<\*itt<<"**\t**";

**else**

cout<<symbols[\*itt]<<"**\t**";

printProd(table[make\_pair(\*itnt,\*itt)]);

cout<<"**\n**";

}

cout<<endl;

}

*// Save parsing table to file*

fstream fout;

fout.open("parsing\_table.txt",ios::trunc | ios::out);

int total=t.size()\*nt.size();

fout<<total<<endl;

**for**(itnt = nt.begin(); itnt!=nt.end();itnt++){ *//non terminal loop*

**for**(itt = t.begin(); itt!=t.end();itt++){ *//terminal loop*

fout<<(\*itnt)<<endl;

fout<<(\*itt)<<endl;

fout<<table[make\_pair(\*itnt,\*itt)]<<"**\n**";

}

}

fout<<"S"<<endl;

fout.close();

cout<<"Parsing table written to file**\n**";

}

**return** 0;

}

#include<bits/stdc++.h>

**using** **namespace** std;

map<string,string> symbols;

vector<pair<int,int> >rc;

*// Function to populate the symbol mapping*

void populateSym()

{

fstream file2;

string str;

file2.open("mapping.txt", ios::in);

**while**(getline(file2,str))

{

string first="";

first+=str[0];

symbols.insert(make\_pair(first,str.substr(2)));

}

file2.close();

*// map<string,string>::iterator it;*

*// for(it=symbols.begin();it!=symbols.end();it++)*

*// cout<<it->first<<"\t\t"<<it->second<<endl;*

}

*// Function to print a production*

void printProd(string prod)

{

int i;

*// cout<<prod<<"\t\t\t\t";*

**if**(prod=="pop" || prod=="scan")

**return**;

string actual="";

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

{

string pr="";

pr+=prod[i];

**if**(i==1)

actual+=" -> ";

**else** **if**(symbols.find(pr)==symbols.end())*// Trivial characters*

actual+=pr+" ";

**else**

actual+=symbols[pr]+" ";

}

cout<<actual;

}

void print(vector<pair<string,string> > v,vector<pair<int,int> >rcl){

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

cout<<v[i].first<<"**\t\t**";

**if**(symbols.find(v[i].second)==symbols.end())

cout<<v[i].second<<"**\t\t**";

**else**

cout<<symbols[v[i].second]<<"**\t\t**";

cout<<rcl[i].first<<"**\t\t**"<<rcl[i].second<<endl;

}

}

void printvector(vector<string> a){

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

cout<<a[i]<<endl;

}

}

vector<string> my(vector<string> v,vector<string> & vars,vector<pair<int,int> >rc){

vector<string> mylist;

*//vector<string> vars;*

vector<pair<string,string> > mp;

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

**if**(v[i]=="program"){

mylist.push\_back("p");

mp.push\_back(make\_pair(v[i],"p"));

}

**else** **if**(v[i]=="uses"){

mylist.push\_back("l");

mp.push\_back(make\_pair(v[i],"l"));

}

**else** **if**(v[i]=="real"){

mylist.push\_back("r");

mp.push\_back(make\_pair(v[i],"r"));

}

**else** **if**(v[i]=="integer"){

mylist.push\_back("u");

mp.push\_back(make\_pair(v[i],"u"));

}

**else** **if**(v[i]=="var"){

mylist.push\_back("v");

mp.push\_back(make\_pair(v[i],"v"));

}

**else** **if**(v[i]=="function"){

mylist.push\_back("f");

mp.push\_back(make\_pair(v[i],"f"));

}

**else** **if**(v[i]=="begin"){

mylist.push\_back("b");

mp.push\_back(make\_pair(v[i],"b"));

}

**else** **if**(v[i]=="end"){

mylist.push\_back("e");

mp.push\_back(make\_pair(v[i],"e"));

}

**else** **if**(v[i]=="get"){

mylist.push\_back("g");

mp.push\_back(make\_pair(v[i],"g"));

}

**else** **if**(v[i]=="put"){

mylist.push\_back("q");

mp.push\_back(make\_pair(v[i],"q"));

}

**else** **if**(v[i][0]=='('){

mylist.push\_back("x");

mp.push\_back(make\_pair(v[i],"x"));

}

**else** **if**(v[i]=="const"){

mylist.push\_back("c");

mp.push\_back(make\_pair(v[i],"c"));

}

**else** **if**(v[i]=="?" || v[i]==":"){

mylist.push\_back(v[i]);

mp.push\_back(make\_pair(v[i],v[i]));

}

**else** **if**(v[i]=="." || v[i]==";" || v[i]==","){

mylist.push\_back(v[i]);

mp.push\_back(make\_pair(v[i],v[i]));

}

**else** **if**(v[i]=="==" || v[i]=="=" || v[i]=="<" || v[i]==">" || v[i]=="<=" || v[i]==">="){

**if**(v[i]=="<="){

mylist.push\_back("y");

mp.push\_back(make\_pair(v[i],"y"));

}

**else** **if**(v[i]==">="){

mylist.push\_back("z");

mp.push\_back(make\_pair(v[i],"z"));

}

**else**{

mylist.push\_back(v[i]);

mp.push\_back(make\_pair(v[i],v[i]));

}

}

**else** **if**(v[i]=="+" || v[i]=="-" || v[i]=="\*"){

mylist.push\_back(v[i]);

mp.push\_back(make\_pair(v[i],v[i]));

}

**else** **if**((v[i][0]>='0' && v[i][0]<='9') || (v[i][0]=='-' && v[i][1]>='0' && v[i][1]<='9')){

mylist.push\_back("n");

mp.push\_back(make\_pair(v[i],"n"));

}

**else**{

**if**(v[i]!=""){

mylist.push\_back("i");

mp.push\_back(make\_pair(v[i],"i"));

vars.push\_back(v[i]);

}

}

}

*//cout<<"tokens\t converted\n";*

cout<<"token**\t**converted token**\t**row**\t**column**\n**";

print(mp,rc);

**return** mylist;

}

bool isdelim(char c){

**if**(c==',' || c==';' || c==' ' || c=='\t' || c=='\n' || c=='?' || c==':' || c=='=' || c=='>' || c=='<' || c=='+' || c=='\*')

**return** true;

**return** false;

}

vector<string> extract(vector<string> s, vector<pair<int,int> >&rc){

vector<string> store;

**for**(int j=0;j<s.size();j++){

string p=s[j];

string temp="";

int tab=0;

int tag=0;

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

*//cout<<p[i]<<" ";*

int store\_i=i;

**if**(tag){

store\_i=tab+i;

}

**if**(isdelim(p[i])){

**if**(temp.size()!=0){

store.push\_back(temp);

int len123=temp.size();

rc.push\_back(make\_pair(j+1,store\_i+1-len123));

}

*//cout<<temp<<endl;*

*//while(i<p.length() && p[i]==' ')*

*// i++;*

**if**(p[i]==',' || p[i]==';' || p[i]=='?' || p[i]==':' || p[i]=='.' || p[i]=='=' || p[i]=='+' || p[i]=='-' || p[i]=='\*'){

**if**(p[i]=='.'){

**if**((p[i-1]>='0' && p[i-1]<='9') && (p[i+1]>='0' && p[i+1]<='9'))

temp=temp+string(1,p[i]);

**if**(i==p.length()-1){

store.push\_back(temp);

int len123=temp.size();

rc.push\_back(make\_pair(j+1,store\_i+2-len123));

}

**else**{

store.push\_back(string(1,p[i]));

rc.push\_back(make\_pair(j+1,store\_i+1));

}

}

**else** **if**(p[i]=='-'){

**if**((p[i-1]=='=' || p[i-1]=='<' || p[i-1]=='>') && (p[i+1]>='0' && p[i+1]<='9'))

temp=temp+string(1,p[i]);

**if**(i==p.length()-1){

store.push\_back(temp);

int len123=temp.size();

rc.push\_back(make\_pair(j+1,store\_i+2-len123));

}

**else**{

store.push\_back(string(1,p[i]));

rc.push\_back(make\_pair(j+1,store\_i+1));

}

}

**else**{

store.push\_back(string(1,p[i]));

rc.push\_back(make\_pair(j+1,store\_i+1));

}

}

**else** **if**(p[i]=='\t'){

tab+=3;

tag=1;

}

**else** **if**(p[i]=='<' && p[i+1]=='='){

store.push\_back("<=");

rc.push\_back(make\_pair(j+1,store\_i+1));

i++;

}

**else** **if**(p[i]=='>' && p[i+1]=='='){

store.push\_back(">=");

rc.push\_back(make\_pair(j+1,store\_i+1));

i++;

}

**else** **if**(p[i]=='>'){

store.push\_back(">");

rc.push\_back(make\_pair(j+1,store\_i+1));

}

**else** **if**(p[i]=='<'){

store.push\_back("<");

rc.push\_back(make\_pair(j+1,store\_i+1));

}

temp="";

}

**else**{

**if**(p[i]=='('){

rc.push\_back(make\_pair(j+1,store\_i+1));

string w="";

**while**(p[i]!=')'){

w=w+string(1,p[i]);

i++;

}

w=w+string(1,p[i]);

store.push\_back(w);

}

**else**{

temp=temp+string(1,p[i]);

**if**(i==p.length()-1){

store.push\_back(temp);

int len123=temp.size();

rc.push\_back(make\_pair(j+1,store\_i+2-len123));

*//rc.push\_back(make\_pair(j+1,i+1));*

*//cout<<temp<<endl;*

}

}

}

}

}

**return** store;

}

*// Function to print stack*

void printStack(stack<char> st)

{

stack<char> temp;

string stack="";

**while**(!st.empty())

{

temp.push(st.top());

st.pop();

}

**while**(!temp.empty())

{

st.push(temp.top());

stack+=temp.top();

temp.pop();

}

cout<<"Stack: "<<stack<<endl;

}

*// Function to parse a string*

void parse(map< pair<char, char>, string > table, vector<string> expr, char startSym)

{

*// Create the stack and push $*

stack<char> st;

st.push('$');

*// Push start symbol onto stack*

st.push(startSym);

int i=0,j;

**while**(!st.empty() && i<expr.size())

{

*// First check if appropriate production exists*

pair<char,char> temp;

char ch=expr[i][0];

temp=make\_pair(st.top(),ch);

*// cout<<temp.first<<", "<<temp.second;*

*// Check if there is a match*

**if**(st.top()==ch)

{

cout<<"Action: match, Popping "<<st.top()<<"**\t\t\t\t\t\t\t\t**";

**if**(symbols.find(expr[i])==symbols.end())

cout<<expr[i]<<"**\t\t\t\t**";

**else**

cout<<symbols[expr[i]]<<"**\t\t\t\t**";

printStack(st);

i++;

st.pop();

**continue**;

}

**if**(table.find(temp)==table.end())

{

cout<<"Parse Error"<<endl;

**break**;

}

**else**

**if**(table[temp]=="scan" || table[temp]=="pop")*// If valid production not found then error*

{

cout<<"Parse error at: "<<rc[i].first<<":"<<rc[i].second<<endl;

**if**(table[temp]=="scan")

{

cout<<"Scan"<<endl;

i++;

**continue**;

}

**else** **if**(table[temp]=="pop")

{

cout<<"Pop"<<endl;

**if**(st.top()=='$')

st.push('S');

**else**

st.pop();

printStack(st);

**continue**;

}

}

*// If valid production exists*

string pr=table[temp];

cout<<"Action: Applying **\t\t\t**";

printProd(pr);

*// cout<<pr;*

cout<<"Popping "<<st.top()<<"**\t\t**";

*// if(symbols.find(expr[i])==symbols.end())*

*// cout<<expr[i]<<"\t\t\t\t\t";*

*// else*

*// cout<<symbols[expr[i]]<<"\t\t\t\t\t";*

cout<<expr[i]<<"**\t\t\t\t\t**";

printStack(st);

st.pop();

**if**(pr[2]!='#')

*// push string onto stack*

**for**(j=pr.length()-1;j>=2;j--)

st.push(pr[j]);

}

}

int main(int argc, char **const** \*argv[])

{

populateSym();

fstream file;

string word, t, q, filename;

filename = "test.pas";

file.open(filename.c\_str());

vector<string> store;

string str;

**while**(getline(file,str)){

store.push\_back(str.c\_str());

}

printvector(store);

store=extract(store,rc);

vector<string> tokens;

vector<string> vars;

tokens=my(store,vars,rc); *//tokens are stored as per converted rules*

*// ============= Parsing =======================*

cout<<"Parsing**\n**";

*// Create parsing table*

int i,j,num;

char start;

*// cout<<"Enter number of entries in table"<<endl;*

cin>>num;

cout<<num<<endl;

map< pair<char, char>, string > parsingTab;

*// Take input*

*// cout<<"For every entry first line is the non terminal second terminal third the production"<<endl;*

**for**(i=0;i<num;i++)

{

char nonter,ter;

string prod;

cin>>nonter;

cin>>ter;

cin>>prod;

parsingTab[make\_pair(nonter,ter)]=prod;

}

cin>>start;

tokens.push\_back("$");

parse(parsingTab,tokens,start);

**return** 0;

}

1. **Output**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
| The above figures show the parsing table of the top down parser |

**Test Input file 1:**

**program** p1

**uses** a,b,c

**const** k=5,g=0

**var** x,y=integer;

**function** f1

fa,fb=integer;

fc=real;

**begin**

get i;

fc=-56.5;

fb=fb5+fb;

fc=5>3?3:fb;

put fb;

**end**;

**begin**

f1=5;

**end** .

|  |
| --- |
|  |
|  |
| **The above figures show the lexical tokenizing of the program** |
|  |
|  |
|  |
|  |
| **The above figures shows the parsing procedure** |

**Test Input file 2:**

**program** p1

**uses** a,b,c

**const** k=5,g

**var** x,y=integer;

**function** f1

fa,fb=integer;

fc=real;

**begin**

get i;

fc=-56.5;

fb=fb5+fb;

fc=5>3?3:fb;

put fb;

**end**;

**begin**

f15;

**end** .

|  |
| --- |
|  |
|  |
| **The above figures show the lexical tokenizing of the program** |
|  |
| **The above figures shows the parsing procedure** |

**Test Input file 3:**

**program** p1

**uses** a,b,c

**const** k=5,g=0

**var** x,y=nteger;

**function**

fa,fb=integer;

fc=real;

**begin**

get i;

fc=-56.5;

fb=fb5+fb;

fc=5>3?3:fb;

put fb;

**end**;

**begin**

f1=5;

**end** .

|  |
| --- |
|  |
|  |
| **The above figures show the lexical tokenizing of the program** |
|  |
|  |
|  |
|  |
| **The above figures shows the parsing procedure** |