JADAVPUR UNIVERSITY

COMPILER DESIGN

LEXICAL ANALYSER &

LL 1 PARSER

ABHIJIT DUTTA

001410501004

B.CSE 3RD YEAR 2ST SEM

GROUP A1

CONTENTS

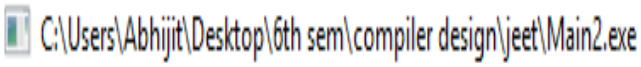
Grammar

mygrammar.txt - Notepad

File Edit Format View Help

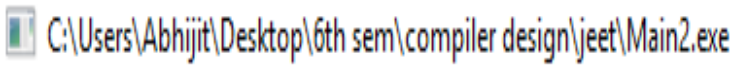
```
<program>       ::=     "main" "(" ")" "{" <statements> "}"
<datatypes>     ::=     "int" | "float" | "void"
<block> ::=     "{" <statements> "}"
<statements>    ::=     <statements> <astatement> | <astatement>
<astatement>    ::=     <declaration> ";" | <assignment> ";" | <ifstruct> | <forstruct> | <readstruct> | <writestruct>
<declaration>   ::=     <datatypes> "identifiers"
<assignment>    ::=     "identifiers" "=" <expression>
<ifstruct>      ::=     "if" "(" <expression> ")" <block> <elsestruct>
<elsestruct>    ::=     "else" <block> | Epsilon
<forstruct>     ::=     "for" "(" "identifiers" "=" <expression> ";" "identifiers" "<" <expression> ";" "identifiers" "=" <expression> ")" <block>
<expression>    ::=     <subexpression>
<relop> ::=     "<" | ">"
<subexpression> ::=     <subexpression> <additive> <term> | <term>
<additive>      ::=     "+" | "-"
<term>  ::=     <term> <multiplicative> <factor> | <term> <relop> <factor> | <factor>
<multiplicative>        ::=     "*" | "/"
<factor>        ::=     "(" <subexpression> ")" | "identifiers" | "numbers"
<readstruct>    ::=     "read" "(" "identifiers" ")" ";"
<writestruct>   ::=     "write" "(" "identifiers" ")" ";"
```

Grammar after removal of Left Recursion




```
------Grammar------
<additive> ->  "+" | "-"
<assignment> ->  "identifiers""="<expression>
<astatement> ->  <declaration>";" | <assignment>";" | <ifstruct> | <forstruct> | <readstruct> | <writestruct>
<block> ->  "{"<statements>"}"
<datatypes> ->  "int" | "float" | "void"
<declaration> ->  <datatypes>"identifiers"
<elsestruct> ->  "else"<block> | Epsilon
<expression> ->  <subexpression>
<factor> ->  "("<subexpression>")" | "identifiers" | "numbers"
<forstruct> ->  "for""("identifiers""="<expression>";""identifiers""<"<expression>";""identifiers""="<expression>")"<block>
<ifstruct> ->  "if""("<expression>")"<block><elsestruct>
<multiplicative> ->  "*" | "/"
<program> ->  "main""("")""{"<statements>"}"
<readstruct> ->  "read""("identifiers"")"";"
<relop> ->  "<" | ">"
<statements'> ->  <astatement><statements'> | Epsilon
<statements> ->  <astatement><statements'>
<subexpression'> ->  <additive><term><subexpression'> | Epsilon
<subexpression> ->  <term><subexpression'>
<term'> ->  <multiplicative><factor><term'> | <relop><factor><term'> | Epsilon
<term> ->  <factor><term'>
<writestruct> ->  "write""("identifiers"")"";"
```

First Pos




```
------First------
<additive> ->  "+" | "-"
<assignment> ->  "identifiers"
<astatement> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write"
<block> ->  "{"
<datatypes> ->  "float" | "int" | "void"
<declaration> ->  "float" | "int" | "void"
<elsestruct> ->  "else" | Epsilon
<expression> ->  "(" | "identifiers" | "numbers"
<factor> ->  "(" | "identifiers" | "numbers"
<forstruct> ->  "for"
<ifstruct> ->  "if"
<multiplicative> ->  "*" | "/"
<program> ->  "main"
<readstruct> ->  "read"
<relop> ->  "<" | ">"
<statements'> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write" | Epsilon
<statements> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write"
<subexpression'> ->  "+" | "-" | Epsilon
<subexpression> ->  "(" | "identifiers" | "numbers"
<term'> ->  "*" | "/" | "<" | ">" | Epsilon
<term> ->  "(" | "identifiers" | "numbers"
<writestruct> ->  "write"
```

Follow Pos

C:\Users\Abhijit\Desktop\6th sem\compiler design\jeet\Main2.exe

```
------Follow------
<additive> ->  "(" | "identifiers" | "numbers"
<assignment> ->  ";"
<astatement> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write" | Epsilon
<block> ->  "else" | "for" | Epsilon
<datatypes> ->  "identifiers"
<declaration> ->  ";"
<elsestruct> ->  "if"
<expression> ->  ")" | ";" | "identifiers"
<factor> ->  "*" | "/" | "<" | ">" | Epsilon
<forstruct> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write"
<ifstruct> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write"
<multiplicative> ->  "(" | "identifiers" | "numbers"
<readstruct> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write"
<relop> ->  "(" | "identifiers" | "numbers"
<statements'> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write" | Epsilon
<statements> ->  "}"
<subexpression'> ->  "(" | "+" | "-" | "identifiers" | "numbers" | Epsilon
<subexpression> ->  "(" | ")" | "identifiers" | "numbers"
<term'> ->  "(" | "*" | "/" | "<" | ">" | "identifiers" | "numbers" | Epsilon
<term> ->  "+" | "-" | Epsilon
<writestruct> ->  "float" | "for" | "identifiers" | "if" | "int" | "read" | "void" | "write"
```

LL1 - Parsing Table

C:\Users\Abhijit\Desktop\6th sem\compiler design\jeet\Main2.exe

```
Parse Table
<additive>->>   "+":<<additive> , "+">  "-":<<additive> , "-">
<assignment>->> "identifiers":<<assignment> , "identifiers""="<expression>>
<astatement>->> "float":<<astatement> , <writestruct>>  "for":<<astatement> , <writestruct>>    "identifiers":<<astatement> , <writestruct>>    "if":<<astatement> , <writestruct>>    "int":<<astatement> , <writestruct>>
"read":<<astatement> , <writestruct>>   "void":<<astatement> , <writestruct>>   "write":<<astatement> , <writestruct>>
<block>->>      "{":<<block> , "{"<statements>"}">
<datatypes>->>  "float":<<datatypes> , "float"> "int":<<datatypes> , "int">    "void":<<datatypes> , "void">
<declaration>->>        "float":<<declaration> , <datatypes>"identifiers">      "int":<<declaration> , <datatypes>"identifiers">        "void":<<declaration> , <datatypes>"identifiers">
<elsestruct>->> "else":<<elsestruct> , "else"<block>>   "if":<<elsestruct> , Epsilon>   Epsilon:<<elsestruct> , Epsilon>
<expression>->> "(":<<expression> , <subexpression>>    "identifiers":<<expression> , <subexpression>>  "numbers":<<expression> , <subexpression>>
<factor>->>     "(":<<factor> , "("<subexpression>")">  "identifiers":<<factor> , "identifiers">       "numbers":<<factor> , "numbers">
<forstruct>->>  "for":<<forstruct> , "for""("identifiers""="<expression>";"identifiers""<"<expression>";"identifiers""="<expression>")"<block>>
<ifstruct>->>   "if":<<ifstruct> , "if""("<expression>")"<block><elsestruct>>
<multiplicative>->>     "*":<<multiplicative> , "*">    "/":<<multiplicative> , "/">
<program>->>    "main":<<program> , "main""("")""{"<statements>"}">
<readstruct>->> "read":<<readstruct> , "read""("identifiers"")"";">
<relop>->>      "<":<<relop> , "<">     ">":<<relop> , ">">
<statements'>->>        "float":<<statements'> , Epsilon>       "for":<<statements'> , Epsilon> "identifiers":<<statements'> , Epsilon> "if":<<statements'> , Epsilon>  "int":<<statements'> , Epsilon> "read":<<statements'> ,
Epsilon>      "void":<<statements'> , Epsilon>        "write":<<statements'> , Epsilon>       Epsilon:<<statements'> , Epsilon>
<statements>->> "float":<<statements> , <astatement><statements'>>      "for":<<statements> , <astatement><statements'>>        "identifiers":<<statements> , <astatement><statements'>>        "if":<<statements> , <astatement
><statements'>>       "int":<<statements> , <astatement><statements'>>        "read":<<statements> , <astatement><statements'>>       "void":<<statements> , <astatement><statements'>>       "write":<<statements> , <astatemen
t><statements'>>
<subexpression'>->>     "(":<<subexpression'> , Epsilon>        "+":<<subexpression'> , Epsilon>        "-":<<subexpression'> , Epsilon>        "identifiers":<<subexpression'> , Epsilon>      "numbers":<<subexpression'> , Ep
silon>        Epsilon:<<subexpression'> , Epsilon>
<subexpression>->>      "(":<<subexpression> , <term><subexpression'>>  "identifiers":<<subexpression> , <term><subexpression'>>        "numbers":<<subexpression> , <term><subexpression'>>
<term'>->>      "(":<<term'> , Epsilon> "*":<<term'> , Epsilon> "/":<<term'> , Epsilon> "<":<<term'> , Epsilon> ">":<<term'> , Epsilon> "identifiers":<<term'> , Epsilon>       "numbers":<<term'> , Epsilon>   Epsilon:<<term'>
 , Epsilon>
<term>->>       "(":<<term> , <factor><term'>>  "identifiers":<<term> , <factor><term'>>        "numbers":<<term> , <factor><term'>>
<writestruct>->>        "write":<<writestruct> , "write""("identifiers"")"";">
```

MAIN CODE

MainCode.cpp

```
#include<bits/stdc++.h>
using namespace std;

set<string> terminals,non_terminals;
map <string , list <string> > grammar;
map <string , set <string> > first,follow;
map <string , map <string, pair<string,string> > > table;

pair<string,int> tokenize(string s,int pos,char ch)
{
  int i,flag=0;
  string str;
  for(i=pos; (s[i] != ch || flag)  && i<s.length(); i++)
  {
        if(s[i]=='/' && s[i+1] =='/')       return make_pair(str,-1);
        if(s[i]!='\t' && s[i]!=' ')
              str+=s[i];
        if(s[i]=='"')             flag^=1;
  }
  return make_pair(str,i+1);
}

void print(map<string,set <string> > code)
{
  for(map<string,set <string> > ::iterator i=code.begin();i!=code.end();i++)
  {
        cout<<i->first<<" ->  ";
        for(set <string> ::iterator j=(i->second).begin(); j!=(i->second).end() && j!=--(i->second).end(); j++)
        {
              cout<<*j<<" | ";
        }
        cout<<*(--(i->second).end());
        cout<<endl;
  }
}

void print(map<string,list <string> > code)
{
  for(map<string,list <string> > ::iterator i=code.begin();i!=code.end();i++)
  {
        cout<<i->first<<" ->  ";
        for(list <string> ::iterator j=(i->second).begin(); j!=(i->second).end() && j!=--(i->second).end(); j++)
        {
              cout<<*j<<" | ";
```

```
            }
            cout<<*(--(i->second).end());
            cout<<endl;
  }
}

void print(map<string,map<string,pair<string,string> > > table)
{
  for(map<string,map<string, pair<string,string> > >::iterator i=table.begin();i!=table.end();i++)
  {
            cout<<(i)->first<<"->>          ";
            for(map<string,pair<string,string> > ::iterator j=(i->second).begin();j!=(i->second).end();j++)
            {
                        cout<<j->first<<":<"<<(j->second).first<<" , "<<(j->second).second<<">          ";
            }
            cout<<endl;
  }
}

pair<string, bool> checkLR(string b, string a)                                    //Check left Recursive
{
  pair<string, bool> temp;
  temp = make_pair("", false);
  string c = b.substr(0, a.size());
  if(c == a)
  {
            temp.second = true;
            temp.first = b.substr(a.size(), b.size()- a.size());
  }
  return temp;
}

void  evaluateFirst(string non_terminal,set<string> terminals,set<string> non_terminals)    //Evaluate First
{
  string str;
  vector<string> s;
  s.push_back(non_terminal);
  while(!s.empty())
  {
            string token=s.back();
            s.pop_back();
            for(list<string>::iterator itr=grammar[token].begin();itr!=grammar[token].end();itr++)
            {
                        if(token == non_terminal)
                        {
                                    str=*itr;
                        }
                        for(int i=1;i<=(*itr).length();i++)
                        {
                                    string temp=(*itr).substr(0,i);
                                    if(non_terminals.find(temp)!=non_terminals.end())
```

```
                {
                        s.push_back(temp);break;
                }
                if(terminals.find(temp)!=terminals.end())
                {
                        first[non_terminal].insert(temp);
                        table[non_terminal][temp]=make_pair(non_terminal,str);
                        break;
                }
        }
    }
  }
}

void evaluateFollow(set<string> terminals,set<string> non_terminals)                    //Evaluates Follow Pos
{
  for(map<string, list<string> >::iterator itr=grammar.begin();itr!=grammar.end();itr++)
  {
        for(list<string> ::iterator ptr=itr->second.begin();ptr!=itr->second.end();ptr++)
        {
                for(int i=(*ptr).length()-1;i>0;)
                {
                        bool jflag=false;
                        for(int j=i;j>=0 && !jflag;j--)
                        {
                                if(terminals.find((*ptr).substr(j,i-j+1))!=terminals.end())
                                {
                                        for(int k=j;k>=0;k--)
                                        {
                                                if(non_terminals.find((*ptr).substr(k,j-k))!=non_terminals.end())
                                                {
                                                        follow[(*ptr).substr(k,j-k)].insert((*ptr).substr(j,i-j+1));
                                                        i=j-1;
                                                        jflag=true;
                                                        break;
                                                }
                                                else if(terminals.find((*ptr).substr(k,j-k))!=terminals.end())
                                                {
                                                        i=j-1;
                                                        jflag=true;
                                                        break;
                                                }
                                                else if(k==0)
                                                {
                                                        i=j-1;
                                                        jflag=true;
                                                }
                                        }
                                }
                                if(i==(*ptr).length()-1 && non_terminals.find((*ptr).substr(j,i-j+1))!=non_terminals.end())
                                {
```

```
                                        for(set<string> ::iterator it=first[itr->first].begin();it!=first[itr->first].end();it++)
                                        {
                                                follow[(*ptr).substr(j,i-j+1)].insert(*it);
                                        }
                                }
                                if(non_terminals.find((*ptr).substr(j,i-j+1))!=non_terminals.end())
                                {
                                        for(int k=j;k>=0;k--)
                                        {
                                                if(terminals.find((*ptr).substr(k,j-k))!=terminals.end())
                                                {
                                                        i=j-1;
                                                        jflag=true;
                                                        break;
                                                }
                                                else if(non_terminals.find((*ptr).substr(k,j-k))!=non_terminals.end())
                                                {
                                                        for(set<string> ::iterator it=first[(*ptr).substr(j,i-
j+1)].begin();it!=first[(*ptr).substr(j,i-j+1)].end();it++)
                                                        {
                                                                follow[(*ptr).substr(k,j-k)].insert(*it);
                                                        }
                                                        i=j-1;
                                                        jflag=true;
                                                        break;
                                                }
                                                else if(k==0)
                                                {
                                                        i=j-1;
                                                        jflag=true;
                                                }
                                        }
                                }
                        }
                }
        }
  }
}

int main()
{
  //extract grammar from file
  fstream fin;
  string line;
  fin.open("mygrammar.txt",ios::in);
  while(getline(fin,line))
  {
        int i=0;
        pair<string,int> terminal=tokenize(line,i,':');
        i=terminal.second;
        if(i== -1)  continue;
```

```
		i+=2;
		while(i<line.length())
		{
			pair <string,int> tokens=tokenize(line,i,'|');
			grammar[terminal.first].push_back(tokens.first);
			if(tokens.second== -1)		break;
			i=tokens.second;
		}
	}

	//remove left recursion
	for(map<string,list<string> > ::iterator itr=grammar.begin();itr!=grammar.end();itr++)
	{
		bool flag=false;
		for(list<string> ::iterator i=itr->second.begin();i!=itr->second.end();i++)
		{
			pair<string,bool> token=checkLR(*i,itr->first);
			if(token.second==true)
			{
				flag=true;
				grammar[(itr->first).substr(0,(itr->first).size()-1) +"'>"].push_back(token.first+(itr->first).substr(0,(itr-
>first).size()-1) +"'>");
			}
		}
		if(flag)
		{
			grammar[(itr->first).substr(0,(itr->first).size()-1) +"'>"].push_back("Epsilon");
			for(list<string> ::iterator i=itr->second.begin();i!=itr->second.end();i++)
			{
				pair<string,bool> token=checkLR(*i,itr->first);
				if(token.second==false)
				{
					list<string> ::iterator it=++i;--i;
					replace(i,it,*i,(*i)+ (itr->first).substr(0,(itr->first).size()-1) +"'>");
				}
			}
			for(list<string> ::iterator i=itr->second.begin();i!=itr->second.end();)
			{
				pair<string,bool> token=checkLR(*i,itr->first);
				if(token.second==true)
				{
					itr->second.erase(i++);
				}
				else i++;
			}
		}
	}
	cout<<"\n------Grammar------\n";print(grammar);

	//generate first follow
	for(map<string,list<string> > ::iterator itr=grammar.begin();itr!=grammar.end();itr++)
```

```
{
		non_terminals.insert(itr->first);
}
for(map<string,list<string> > ::iterator itr=grammar.begin();itr!=grammar.end();itr++)
{
		for(list<string> ::iterator i=itr->second.begin();i!=itr->second.end();i++)
		{
				bool flag=false;
				string temp;
				for(int j=0;j<(*i).size();j++)
				{
						if((*i)[j]=='"')		flag=!flag;
						if(flag)	temp+=(*i)[j];
						else
						{
								terminals.insert(temp+'"');
								temp.clear();
						}
				}
		}
}
terminals.erase("\"");
terminals.insert("Epsilon");
for(map<string,list<string> > ::iterator itr=grammar.begin();itr!=grammar.end();itr++)
{
		for(list<string> ::iterator i=itr->second.begin();i!=itr->second.end();i++)
		{
				if(terminals.find(*i)!=terminals.end())	first[itr->first].insert(*i);
		}
}
for(map<string,list<string> > ::iterator itr=grammar.begin();itr!=grammar.end();itr++)
{
		evaluateFirst(itr->first,terminals,non_terminals);
}
cout<<"\n------First------\n";print(first);
evaluateFollow(terminals,non_terminals);
cout<<"\n------Follow------\n";print(follow);

//generate LL1 parse table
for(map<string,set<string> > ::iterator itr=first.begin();itr!=first.end();itr++)
{
		for(set<string> ::iterator ptr= (itr->second).begin();ptr!=(itr->second).end();ptr++)
		{
				if(*ptr == "Epsilon")
				{
						for(set <string> :: iterator jtr= follow[itr->first].begin();jtr != follow[itr->first].end();jtr++)
						{
								table[itr->first][*jtr]=make_pair(itr->first,"Epsilon");
						}
				}
		}
```

```
}
cout<<"\n\nParse Table\n";print(table);

//take tokenized C code
fstream file2;
file2.open("output_program");
ofstream f3;
f3.open("output");

std::vector<string> program;

while ( getline (file2,str) )
{
	program.push_back(str);
}

//parse the C code

program.push_back("$");
vector<string> stck;
stck.push_back("$");
stck.push_back(non_terminals[0]);
int i=0, err = 0;
while(i < production.size() && stck.size() > 0)
{
	pair<string, int> term = make_pair("", 0);
	term = tokenize(program[i], term.second, ' ');
	do
	{
		f3 << "\nTOKEN = "<< term.first << " " << endl;
		string y = stck.back();
		f3 << "TOP OF STACK = " << y << endl;
		stck.pop_back();
		if(check_terminal(y) || y[0] == '$')
		{
			if(y == term.first)
			{
				term = tokenize(program[i], term.second, ' ');
				if(term.second == -2 )
				{
					break;
				}
				continue;
			}
			else
			{
				err=1;
				cout<<"Error\n";
				break;
			}
```

```
                }
                int j,k;
                for( k=0; k<terminals.size(); k++)
                        if(terminals[k] == term.first)
                                break;
                for( j=0; j<non_terminals.size(); j++)
                        if(non_terminals[j] == y)
                                break;
                string z = parser_table[j][k];
                if(z == " ")
                {
                        err=1;
                        break;
                }
                if(z == "Epsilon")
                        continue;
                string ss="";
                std::vector<string> temp;
                for(j= 0; j < z.size(); j++)
                {
                        if(z[j] == ' ')
                        {
                                temp.push_back(ss);
                                ss = "";
                        }
                        else
                                ss += z[j];
                }
                temp.push_back(ss);
                for(j= temp.size()-1; j>=0; j--)
                {
                        stck.push_back(temp[j]);
                }
        }while(term.second != -2);
        i++;
        if(err)
                break;
  }

  if(!err)
        f3<<"\nINPUT C PROGRAM IS CORRECT\n";
  else
        f3<<"\nINPUT C PROGRAM IS INCORRECT\n";

  //close all open files
  file.close();
  file2.close();
  cout<<"\n\n-------finish----------";
  return 0;
}
```

Tokenizer.c

```
#include <stdio.h>
#include <string.h>

//Storing the tokens
struct map
{
	char type[100], lexme[100];
	int id, x, y;
}TOKENS[500];

//Symbol table
struct symbol_table
{
	char type[100], lexme[100], dtype[100];
	int sc, x,y, ref_x, ref_y;
}SYM_TAB[500];

//Global variables to store scope and indexes to array
int count=-1, scope = 0, ctr = 0;
char prev_id[100];

//Declaring the keywords and operators globally
char keyword[][100] = {
	{ "else" }, { "if" } , {"for"}, {"int"}, {"float"}, {"void"}, {"return"}
};

char arop[][2] = { {"+"}, {"-"}, {"*"}, {"/"} };
char relop[][2] = { {"<"}, {">"} };
char punc[][2] = { {";"},  {","},  {"("},  {")"},  {"{"},  {"}"} };

//Function to check if a string is a keyword
int check_keyword(char token[])
{
	for(int i=0; i<7; i++)
	{
		if(strcmp(token, keyword[i]) == 0)
			return 1;
	}
	return 0;
}

//Function to check if a string is an arithmetic operator
int check_arop(char token[])
{
	for(int i=0; i<4; i++)
	{
		if(strcmp(token, arop[i]) == 0)
			return 1;
```

```
	}
	return 0;
}

//Function to check if a string is a relational operator
int check_relop(char token[])
{
	for(int i=0; i<2; i++)
	{
		if(strcmp(token, relop[i]) == 0)
			return 1;
	}
	return 0;
}

//Function to check if a string is a punctuator
int check_punc(char token[])
{
	for(int i=0; i<6; i++)
	{
		if(strcmp(token, punc[i]) == 0)
			return 1;
	}
	return 0;
}

//Function to check if a string is a constant
int check_const(char token[])
{
	int len = strlen(token);
	for(int i=0; i<len; i++)
	{
		if( !(token[i] >= '0' && token[i] <= '9'))
			return 0;
	}
	return 1;
}

//Function to check the type of token and store it
void tokenize(char token[], int line, int col, FILE *fp3)
{
	count++;
	TOKENS[count].id = (count+1)*5;
	strcpy( TOKENS[count].lexme, token );
	TOKENS[count].x = line;
	TOKENS[count].y = col;

	if(check_keyword(token))
	{
		strcpy(TOKENS[count].type,"Keyword");
		fprintf(fp3,"%s ", token);
```

```
}
else if(token[0] == '=')
{
		strcpy(TOKENS[count].type,"Assignment OP");
		fprintf(fp3,"%s ", token);
}
else if(check_arop(token))
{
		strcpy(TOKENS[count].type,"Arithmetic OP");
		fprintf(fp3,"%s ", token);
}

else if(check_relop(token))
{
		strcpy(TOKENS[count].type,"Relational OP");
		fprintf(fp3,"%s ", token);
}

else if(check_punc(token))
{
		strcpy(TOKENS[count].type,"Punctuator");
		fprintf(fp3,"%s ", token);
}
else if(check_const(token))
{
		strcpy(TOKENS[count].type,"Constant");
		fprintf(fp3,"numbers ");
}
else
{
		strcpy(TOKENS[count].type,"Identifier");
		if(strcmp(token, "main")==0)
			fprintf(fp3,"main ");
		else if(strcmp(token, "read")==0)
			fprintf(fp3,"read ");
		else if(strcmp(token, "write")==0)
			fprintf(fp3,"write ");
		else
		{
			fprintf(fp3,"identifiers ");
			if(strcmp(prev_id, "int") == 0 || strcmp(prev_id, "float") == 0)
			{
				strcpy(SYM_TAB[ctr].type, "Identifier");
				strcpy(SYM_TAB[ctr].lexme, token);
				strcpy(SYM_TAB[ctr].dtype, prev_id);
				SYM_TAB[ctr].x = line;
				SYM_TAB[ctr].y = col;
				SYM_TAB[ctr].sc = scope;
				ctr++;
			}
			else
```

```
                {
                        int err = 1;
                        for(int i=ctr; i>=0; i--)
                        {
                                if(strcmp(token, SYM_TAB[i].lexme) == 0 && SYM_TAB[i].sc <= scope)
                                {
                                        SYM_TAB[i].ref_x = line;
                                        SYM_TAB[i].ref_y = col;
                                        err = 0;
                                        break;
                                }
                        }
                        // Variable out of scope
                        if(err)
                        {
                                printf("ERROR IN CODE. VARIABLE %s OUT OF SCOPE\n", token);
                        }

                }
        }
    }
}

//Function to write output to file
void display()
{
    FILE *fp2 = fopen("tokens", "w");

    //Writing output to file
    fprintf(fp2, "\nTotal tokens = %d\n\n", count+1);
    fprintf(fp2, "TOKEN ID\t  TOKEN TYPE\tLEXME\t POSITION\n");
    for(int i=0; i<=count; i++)
    {
        fprintf(fp2, "%3d %20s\t%5s\t   (%d, %d)\n", TOKENS[i].id, TOKENS[i].type, TOKENS[i].lexme, TOKENS[i].x,
TOKENS[i].y);
    }
    fclose(fp2);
}

//Function to display symbol table after every '}' is encountered
void display_symtab(FILE *fp4)
{
    fprintf(fp4, "CURRENT SCOPE = %d\n", scope);
    for(int i=0; i< ctr; i++)
    {
        if(SYM_TAB[i].sc != 1000)
        {
            fprintf(fp4, "\nName = %s\nType = %s\nData type = %s\nScope = %d\nPosition = %d, %d\nReferenced in :
%d, %d\n",
            SYM_TAB[i].lexme, SYM_TAB[i].type,SYM_TAB[i].dtype ,SYM_TAB[i].sc, SYM_TAB[i].x, SYM_TAB[i].y,
            SYM_TAB[i].ref_x, SYM_TAB[i].y);
```

```
                }
                if(SYM_TAB[i].sc == scope)
                        SYM_TAB[i].sc = 1000;
        }
        fprintf(fp4, "\n\n");
}

//main function
int main()
{
        //Opening input file
        FILE *fp = fopen("input.c", "r");
        FILE *fp3 = fopen("output_prog", "w");
        FILE *fp4 = fopen("symbol_table", "w");
        char c, buff[100], g;

        int i=0, line=1, col=0, flag = 0, comnt = 0;

        while(1)
        {
                if(!flag)
                {
                        c = fgetc(fp);

                        //If end of file then return
                        if(c == EOF)
                                break;
                }
                else
                        c = g;

                col++;

                //If there is a space or new line then tokenize the string found till here
                if(c == ' ' || c == '\n' || c == '\t')
                {
                        if( i>0)
                        {
                                buff[i] = '\0';
                                tokenize(buff, line, col, fp3);
                                strcpy(prev_id, buff);
                                i=0;
                        }
                        if(c == '\n')
                        {
                                line++;
                                col=0;
                                fprintf(fp3,"\n");
                        }
                        continue;
                }
```

```
//Checking for comments
if(c == '/')
{
   g = fgetc(fp);

   //Removing single line comment
            if(g == '/')
            {
               do
               {
                        c = fgetc(fp);
               }while(c != '\n');
               flag = 0;
               comnt = 1;
            }

            //Removing multiple lines comment
            else if(g == '*')
            {
                        do
                        {
                                    c = fgetc(fp);
                                    g = fgetc(fp);
                        }while( c!= '*' && g != '/');
                        flag = 0;
                        comnt=1;
            }

            //If not a comment then continue
            else
            {
                        flag = 1;
                        comnt = 0;
            }
}

char temp[3];
temp[0]=c;

if(!comnt)
{
            //Checking for operator or punctuator
            if(check_punc(temp) || check_relop(temp) || check_arop(temp) || temp[0] == '=')
            {
                        //Tokenizing the string found before the operator
                        if(i>0)
                        {
                                    buff[i] = '\0';
                                    tokenize(buff, line, col, fp3);
                                    col++;
```

```
                    strcpy(prev_id, buff);
                }

                //Tokenizing the operator
                tokenize(temp, line, col, fp3);
                strcpy(prev_id, buff);

                //increasing the scope
                if(temp[0] == '{')
                    scope++;

                //decreasing the scope as well as displaying the symbol table
                else if(temp[0] == '}')
                {
                    display_symtab(fp4);
                    scope--;
                }
                i=0;
                continue;
            }

            //Checking for string literal
            if(c == '"')
            {
                do
                {
                    buff[i++] = c;
                    c = fgetc(fp);
                }while(c!= '"');

                //Tokenizing the string literal
                buff[i++]=c;
                buff[i]='\0';
                count++;
                TOKENS[count].id = (count+1)*5;
                strcpy( TOKENS[count].lexme, buff );
                TOKENS[count].x = line;
                TOKENS[count].y = col;
                strcpy(TOKENS[count].type,"String Literal");
                i=0;
                strcpy(prev_id, "string");
                continue;

            }
            buff[i++] =  c;

        }
        comnt=0;
}

//Write the output in file
```

```
        display();

        fclose(fp);
        fclose(fp3);
        fclose(fp4);
        return 0;
}
```

Input Code

```
int main(void)
{
	int a ;
	int b ;
	read ( a ) ;
	read ( b ) ;
	for ( a = 1; a < 10 ; a = a +1 )
	{
		b = b +1 ;
		int c ;
		if ( a > b )
		{
			c = a -1 ;
		}
		else
		{
			c = a +1;
		}
	}
	write ( b ) ;
}
```

Stack Output

TOKEN = main

TOP OF STACK = program

TOKEN = main

TOP OF STACK = main

TOKEN = (

TOP OF STACK = (

TOKEN =)

TOP OF STACK =)

TOKEN = {

TOP OF STACK = {

TOKEN = int

TOP OF STACK = statements

TOKEN = int

TOP OF STACK = statements'

TOKEN = int

TOP OF STACK = astatement

TOKEN = int

TOP OF STACK = declaration

TOKEN = int

TOP OF STACK = datatypes

TOKEN = int

TOP OF STACK = int

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = ;

TOP OF STACK = ;

TOKEN = int

TOP OF STACK = statements'

TOKEN = int

TOP OF STACK = astatement

TOKEN = int

TOP OF STACK = declaration

TOKEN = int

TOP OF STACK = datatypes

TOKEN = int

TOP OF STACK = int

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = ;

TOP OF STACK = ;

TOKEN = read

TOP OF STACK = statements'

TOKEN = read

TOP OF STACK = astatement

TOKEN = read

TOP OF STACK = readstruct

TOKEN = read

TOP OF STACK = read

TOKEN = (

TOP OF STACK = (

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN =)

TOP OF STACK =)

TOKEN = ;

TOP OF STACK = ;

TOKEN = read

TOP OF STACK = statements'

TOKEN = read

TOP OF STACK = astatement

TOKEN = read

TOP OF STACK = readstruct

TOKEN = read

TOP OF STACK = read

TOKEN = (

TOP OF STACK = (

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN =)

TOP OF STACK =)

TOKEN = ;

TOP OF STACK = ;

TOKEN = for

TOP OF STACK = statements'

TOKEN = for

TOP OF STACK = astatement

TOKEN = for

TOP OF STACK = forstruct

TOKEN = for

TOP OF STACK = for

TOKEN = (

TOP OF STACK = (

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = =

TOP OF STACK = =

TOKEN = numbers

TOP OF STACK = expression

TOKEN = numbers

TOP OF STACK = subexpression

TOKEN = numbers

TOP OF STACK = term

TOKEN = numbers

TOP OF STACK = factor

TOKEN = numbers

TOP OF STACK = numbers

TOKEN = ;

TOP OF STACK = term'

TOKEN = ;

TOP OF STACK = subexpression'

TOKEN = ;

TOP OF STACK = ;

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = <

TOP OF STACK = <

TOKEN = numbers

TOP OF STACK = expression

TOKEN = numbers

TOP OF STACK = subexpression

TOKEN = numbers

TOP OF STACK = term

TOKEN = numbers

TOP OF STACK = factor

TOKEN = numbers

TOP OF STACK = numbers

TOKEN = ;

TOP OF STACK = term'

TOKEN = ;

TOP OF STACK = subexpression'

TOKEN = ;

TOP OF STACK = ;

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = =

TOP OF STACK = =

TOKEN = identifiers

TOP OF STACK = expression

TOKEN = identifiers

TOP OF STACK = subexpression

TOKEN = identifiers

TOP OF STACK = term

TOKEN = identifiers

TOP OF STACK = factor

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = +

TOP OF STACK = term'

TOKEN = +

TOP OF STACK = subexpression'

TOKEN = +

TOP OF STACK = additive

TOKEN = +

TOP OF STACK = +

TOKEN = numbers

TOP OF STACK = term

TOKEN = numbers

TOP OF STACK = factor

TOKEN = numbers

TOP OF STACK = numbers

TOKEN =)

TOP OF STACK = term'

TOKEN =)

TOP OF STACK = subexpression'

TOKEN =)

TOP OF STACK =)

TOKEN = {

TOP OF STACK = block

TOKEN = {

TOP OF STACK = {

TOKEN = identifiers

TOP OF STACK = statements

TOKEN = identifiers

TOP OF STACK = statements'

TOKEN = identifiers

TOP OF STACK = astatement

TOKEN = identifiers

TOP OF STACK = assignment

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = =

TOP OF STACK = =

TOKEN = identifiers

TOP OF STACK = expression

TOKEN = identifiers

TOP OF STACK = subexpression

TOKEN = identifiers

TOP OF STACK = term

TOKEN = identifiers

TOP OF STACK = factor

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = +

TOP OF STACK = term'

TOKEN = +

TOP OF STACK = subexpression'

TOKEN = +

TOP OF STACK = additive

TOKEN = +

TOP OF STACK = +

TOKEN = numbers

TOP OF STACK = term

TOKEN = numbers

TOP OF STACK = factor

TOKEN = numbers

TOP OF STACK = numbers

TOKEN = ;

TOP OF STACK = term'

TOKEN = ;

TOP OF STACK = subexpression'

TOKEN = ;

TOP OF STACK = ;

TOKEN = int

TOP OF STACK = statements'

TOKEN = int

TOP OF STACK = astatement

TOKEN = int

TOP OF STACK = declaration

TOKEN = int

TOP OF STACK = datatypes

TOKEN = int

TOP OF STACK = int

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = ;

TOP OF STACK = ;

TOKEN = if

TOP OF STACK = statements'

TOKEN = if

TOP OF STACK = astatement

TOKEN = if

TOP OF STACK = ifstruct

TOKEN = if

TOP OF STACK = if

TOKEN = (

TOP OF STACK = (

TOKEN = identifiers

TOP OF STACK = expression

TOKEN = identifiers

TOP OF STACK = subexpression

TOKEN = identifiers

TOP OF STACK = term

TOKEN = identifiers

TOP OF STACK = factor

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = >

TOP OF STACK = term'

TOKEN = >

TOP OF STACK = relop

TOKEN = >

TOP OF STACK = >

TOKEN = identifiers

TOP OF STACK = factor

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN =)

TOP OF STACK = term'

TOKEN =)

TOP OF STACK = subexpression'

TOKEN =)

TOP OF STACK =)

TOKEN = {

TOP OF STACK = block

TOKEN = {

TOP OF STACK = {

TOKEN = identifiers

TOP OF STACK = statements

TOKEN = identifiers

TOP OF STACK = statements'

TOKEN = identifiers

TOP OF STACK = astatement

TOKEN = identifiers

TOP OF STACK = assignment

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = =

TOP OF STACK = =

TOKEN = identifiers

TOP OF STACK = expression

TOKEN = identifiers

TOP OF STACK = subexpression

TOKEN = identifiers

TOP OF STACK = term

TOKEN = identifiers

TOP OF STACK = factor

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = -

TOP OF STACK = term'

TOKEN = -

TOP OF STACK = subexpression'

TOKEN = -

TOP OF STACK = additive

TOKEN = -

TOP OF STACK = -

TOKEN = numbers

TOP OF STACK = term

TOKEN = numbers

TOP OF STACK = factor

TOKEN = numbers

TOP OF STACK = numbers

TOKEN = ;

TOP OF STACK = term'

TOKEN = ;

TOP OF STACK = subexpression'

TOKEN = ;

TOP OF STACK = ;

TOKEN = }

TOP OF STACK = statements'

TOKEN = }

TOP OF STACK = }

TOKEN = else

TOP OF STACK = elsestruct

TOKEN = else

TOP OF STACK = else

TOKEN = {

TOP OF STACK = block

TOKEN = {

TOP OF STACK = {

TOKEN = identifiers

TOP OF STACK = statements

TOKEN = identifiers

TOP OF STACK = statements'

TOKEN = identifiers

TOP OF STACK = astatement

TOKEN = identifiers

TOP OF STACK = assignment

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = =

TOP OF STACK = =

TOKEN = identifiers

TOP OF STACK = expression

TOKEN = identifiers

TOP OF STACK = subexpression

TOKEN = identifiers

TOP OF STACK = term

TOKEN = identifiers

TOP OF STACK = factor

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN = +

TOP OF STACK = term'

TOKEN = +

TOP OF STACK = subexpression'

TOKEN = +

TOP OF STACK = additive

TOKEN = +

TOP OF STACK = +

TOKEN = numbers

TOP OF STACK = term

TOKEN = numbers

TOP OF STACK = factor

TOKEN = numbers

TOP OF STACK = numbers

TOKEN = ;

TOP OF STACK = term'

TOKEN = ;

TOP OF STACK = subexpression'

TOKEN = ;

TOP OF STACK = ;

TOKEN = }

TOP OF STACK = statements'

TOKEN = }

TOP OF STACK = }

TOKEN = }

TOP OF STACK = statements'

TOKEN = }

TOP OF STACK = }

TOKEN = write

TOP OF STACK = statements'

TOKEN = write

TOP OF STACK = astatement

TOKEN = write

TOP OF STACK = writestruct

TOKEN = write

TOP OF STACK = write

TOKEN = (

TOP OF STACK = (

TOKEN = identifiers

TOP OF STACK = identifiers

TOKEN =)

TOP OF STACK =)

TOKEN = ;

TOP OF STACK = ;

TOKEN = }

TOP OF STACK = statements'

TOKEN = }

TOP OF STACK = }

TOKEN = $

TOP OF STACK = $

INPUT C PROGRAM IS CORRECT

Output Code

```
main ( )
{
int identifiers ;
int identifiers ;
read ( identifiers ) ;
read ( identifiers ) ;
for ( identifiers = numbers ; identifiers < numbers ; identifiers = identifiers + numbers )
{
identifiers = identifiers + numbers ;
int identifiers ;
if ( identifiers > identifiers )
{
identifiers = identifiers - numbers ;
}
else
{
identifiers = identifiers + numbers ;
}
}
write ( identifiers ) ;
}
```

Symbol Table

```
CURRENT SCOPE = 3

Name = a
Type = Identifier
Data type = int
Scope = 1
Position = 3, 7
Referenced in : 13, 7

Name = b
Type = Identifier
Data type = int
Scope = 1
Position = 4, 7
Referenced in : 11, 7

Name = c
Type = Identifier
Data type = int
Scope = 2
Position = 10, 8
Referenced in : 13, 8


CURRENT SCOPE = 3

Name = a
Type = Identifier
Data type = int
Scope = 1
Position = 3, 7
Referenced in : 17, 7

Name = b
Type = Identifier
Data type = int
Scope = 1
Position = 4, 7
Referenced in : 11, 7

Name = c
Type = Identifier
Data type = int
Scope = 2
Position = 10, 8
Referenced in : 17, 8
```

```
CURRENT SCOPE = 2

Name = a
Type = Identifier
Data type = int
Scope = 1
Position = 3, 7
Referenced in : 17, 7

Name = b
Type = Identifier
Data type = int
Scope = 1
Position = 4, 7
Referenced in : 11, 7

Name = c
Type = Identifier
Data type = int
Scope = 2
Position = 10, 8
Referenced in : 17, 8


CURRENT SCOPE = 1

Name = a
Type = Identifier
Data type = int
Scope = 1
Position = 3, 7
Referenced in : 17, 7

Name = b
Type = Identifier
Data type = int
Scope = 1
Position = 4, 7
Referenced in : 20, 7
```