SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

Intermediate Code generator in C Language

Gudied by:

Dr. M. Baskar

Submitted By:

ARYAN ATUL (RA2011003010844)

MAYANK RAI (RA201100301871)

ARYAN RAJ (RA2011003010895)

Aim: To Implement Intermediate Code Generator in C language.

ABSTRACT

A compiler is a computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Four phases of the frontend compiler are Lexical phase, Syntax phase, Semantic phase and Intermediate code generation. Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. The syntax tree can be converted into a linear representation. This phase is followed by Intermediate code optimization and then machine code generation. Intermediate codes are machine independent codes, but they are close to machine instructions. The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator. Intermediate code generator eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers. If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. In this project, for assignments and expressions, every variable declared and result of every expression is stored in a temporary variable generated. Conditional and iterative statements are handled by goto statements and labels. Precedence to the operators are given so that the computation is done accordingly. The Intermediate code is generated for if, if-else, for, while and do-while. Three address code is generated for arrays also.

CONTENTS

INTRODUCTION

Intermediate Code Generation

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. A three-address code has at most three address locations to calculate the expression. Hence, the intermediate code generator will divide any expression into sub-expressions and then generate the corresponding code. A three-address code can be represented in two forms : quadruples and triples.

Quadruples : Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result.

Triplets : Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Indirect triplets : This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Lex Script

The script written by us is a program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section
%%
Rules Section
%%
C code section

The C code section contains C statements and functions. These statements contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

Yacc Script

A YACC source program is structurally similar to a LEX one.

declarations
%%
rules
%%
routines

- The declaration section may be empty. if the routines section is omitted, the second %% mark may be omitted.
- Blanks, tabs and newlines are ignored except that they may not appear in names.

Declaration Section may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword % token.
- Declaration of the start symbol using the keyword %start
- C code between %{ and %}

Rules Section

A rule has the form:

```
Nonterminal: sentential form
                    | sentential form
                    ……..
             | sentential form
             ;
```

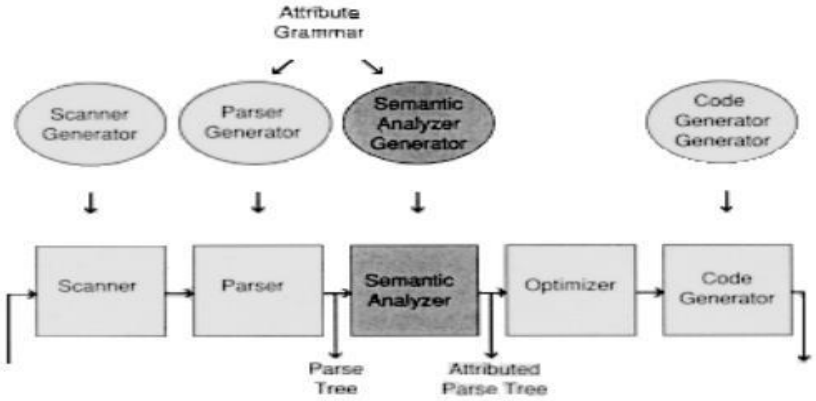
Actions may be associated with rules and are executed when the associated sentential form is matched. Each sentential form has an action associated with it which describes the semantic rules for type mismatch, handling undeclared variables, redeclaration, scope of variables. Also each production has actions which handle the three address code generation.

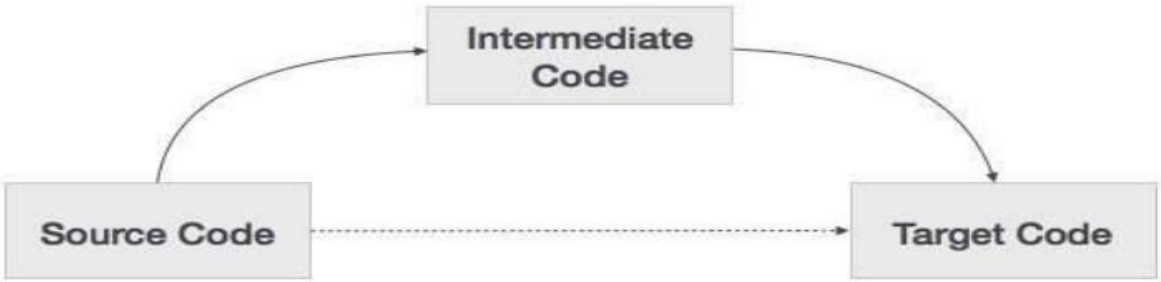
C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned in account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input. The yacc script is run along with the lex script to parse the C code given as input and specify the syntactic and semantic errors (if any) or tell the user that the parsing has been successfully completed. The three address code is generated and the temporary variables generated for the purpose and their values are displayed. Along with this the labels, goto statement and the flow of conditional and iterative statements are also displayed.

Design of Programs

FLow





CODES

LEX Code (parser.l)

```
**************************************************************************************************
%{
int yylineno;
%}

alpha [A-Za-z]
digit [0-9]

%%
[ \t] ;
\n {yylineno++;}
"{"   {scope_start(); return '{';}
"}"   {scope_end(); return '}';}
";"         {  return(';'); }
","         {  return(','); }
":"         {  return(':'); }
"="         {  return('='); }
"("         {  return('('); }
")"         {  return(')'); }
("[")       {  return('['); }
("]")       {  return(']'); }
"."         {  return('.'); }
"&"         {  return('&'); }
"!"         { return('!'); }
"~"         {  return('~'); }
"-"         {  return('-'); }
"+"         {  return('+'); }
"*"         {  return('*'); }
"/"         {  return('/'); }
"%"         {  return('%'); }
"<"         {  return('<'); }
">"         {  return('>'); }
"^"         {  return('^'); }
"|"         {  return('|'); }
"?"         {  return('?'); }
int  {yylval.ival = INT; return INT;}
float  {yylval.ival = FLOAT; return FLOAT;}
void {yylval.ival = VOID; return VOID;}
```

```
else {return ELSE;}
do return DO;
if  return IF;
struct  return STRUCT;
^"#include ".+  return PREPROC;
while  return WHILE;
for  return FOR;
return  return RETURN;
printf  return PRINT;
{alpha}({alpha}|{digit})* {yylval.str=strdup(yytext); return ID;}
{digit}+    {yylval.str=strdup(yytext);return NUM;}
{digit}+\.{digit}+  {yylval.str=strdup(yytext);  return  REAL;}
"<="      return LE;
">="      return  GE;
"=="      return EQ;
"!="   return   NEQ;
"&&"   return   AND;
"||"  return OR;
\/\/.*;
\/\*(.*\n)*.*\*\/ ;
\".*\"   return STRING;
.  return yytext[0];
%%
```

****

YACC Code (parser.y)

****

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "symbolTable.c"

int g_addr = 100;
int i=1,lnum1=0;
int stack[100],index1=0,end[100],arr[10],ct,c,b,fl,top=0,label[20],label_num=0,ltop=0;
char st1[100][10];
char temp_count[2]="0";
int plist[100],flist[100],k=-1,errc=0,j=0;
char temp[2]="t";
char null[2]=" ";
void yyerror(char *s);
int printline();
```

```
extern int yylineno;
void scope_start()
{
  stack[index1]=i;
  i++;
  index1++;
  return;
}
void scope_end()
{
  index1--;
  end[stack[index1]]=1;
  stack[index1]=0;
  return;
}
void if1()
{
  label_num++;
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("\n%s = not %s\n",temp,st1[top]);
        printf("if %s goto L%d\n",temp,label_num);
  temp_count[0]++;
  label[++ltop]=label_num;

}
void if2()
{
  label_num++;
  printf("\ngoto L%d\n",label_num);
  printf("L%d: \n",label[ltop--]);
  label[++ltop]=label_num;
}
void if3()
{
  printf("\nL%d:\n",label[ltop--]);
}
void while1()
{
  label_num++;
  label[++ltop]=label_num;
  printf("\nL%d:\n",label_num);
```

```
}
void while2()
{
  label_num++;
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("\n%s = not %s\n",temp,st1[top--]);
        printf("if %s goto L%d\n",temp,label_num);
  temp_count[0]++;
  label[++ltop]=label_num;
}
void while3()
{
  int y=label[ltop--];
  printf("\ngoto L%d\n",label[ltop--]);
  printf("L%d:\n",y);
}
void dowhile1()
{
  label_num++;
  label[++ltop]=label_num;
  printf("\nL%d:\n",label_num);
}
void dowhile2()
{
        printf("\nif %s goto L%d\n",st1[top--],label[ltop--]);
}
void for1()
{
  label_num++;
  label[++ltop]=label_num;
  printf("\nL%d:\n",label_num);
}
void for2()
{
  label_num++;
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("\n%s = not %s\n",temp,st1[top--]);
        printf("if %s goto L%d\n",temp,label_num);
  temp_count[0]++;
  label[++ltop]=label_num;
```

```
  label_num++;
  printf("goto L%d\n",label_num);
  label[++ltop]=label_num;
  label_num++;
  printf("L%d:\n",label_num);
  label[++ltop]=label_num;
}
void for3()
{
  printf("\ngoto L%d\n",label[ltop-3]);
  printf("L%d:\n",label[ltop-1]);
}
void for4()
{
  printf("\ngoto L%d\n",label[ltop]);
  printf("L%d:\n",label[ltop-2]);
  ltop-=4;
}
void push(char *a)
{
  strcpy(st1[++top],a);
}
void array1()
{
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("\n%s = %s\n",temp,st1[top]);
  strcpy(st1[top],temp);
  temp_count[0]++;
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("%s = %s [ %s ] \n",temp,st1[top-1],st1[top]);
  top--;
  strcpy(st1[top],temp);
  temp_count[0]++;
}
void codegen()
{
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("\n%s = %s %s %s\n",temp,st1[top-2],st1[top-1],st1[top]);
  top-=2;
```

```
  strcpy(st1[top],temp);
  temp_count[0]++;
}
void codegen_umin()
{
  strcpy(temp,"t");
  strcat(temp,temp_count);
  printf("\n%s = -%s\n",temp,st1[top]);
  top--;
  strcpy(st1[top],temp);
  temp_count[0]++;
}
void codegen_assign()
{
  printf("\n%s = %s\n",st1[top-2],st1[top]);
  top-=2;
}
%}

%token<ival> INT FLOAT VOID
%token<str> ID NUM REAL
%token WHILE IF RETURN PREPROC LE STRING PRINT FUNCTION DO ARRAY ELSE STRUCT
STRUCT_VAR FOR GE EQ NE INC DEC AND OR
%left LE GE EQ NEQ AND OR '<' '>'
%right '='
%right UMINUS
%left '+' '-'
%left '*' '/'
%type<str> assignment assignment1 consttype '=' '+' '-' '*' '/' E T F
%type<ival> Type
%union {
        int ival;
        char *str;
  }
%%

start : Function start
  | PREPROC start
  | Declaration start
  |
  ;
```

```
Function : Type ID '(')' CompoundStmt {
  if(strcmp($2,"main")!=0)
  {
        printf("goto F%d\n",lnum1);
  }
  if ($1!=returntype_func(ct))
  {
        printf("\nError : Type mismatch : Line %d\n",printline());
  }

  if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") && strcmp($2,"gets") &&
strcmp($2,"getchar") && strcmp ($2,"puts") && strcmp($2,"putchar") && strcmp($2,"clearerr") &&
strcmp($2,"getw") && strcmp($2,"putw") && strcmp($2,"putc") && strcmp($2,"rewind") &&
strcmp($2,"sprint") && strcmp($2,"sscanf") && strcmp($2,"remove") && strcmp($2,"fflush")))
        printf("Error : Type mismatch in redeclaration of %s : Line %d\n",$2,printline());
  else
  {
        insert($2,FUNCTION);
        insert($2,$1);
        g_addr+=4;
  }
  }
  | Type ID '(' parameter_list ')' CompoundStmt {
  if ($1!=returntype_func(ct))
  {
        printf("\nError : Type mismatch : Line %d\n",printline()); errc++;
  }

  if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") && strcmp($2,"gets") &&
strcmp($2,"getchar") && strcmp ($2,"puts") && strcmp($2,"putchar") && strcmp($2,"clearerr") &&
strcmp($2,"getw") && strcmp($2,"putw") && strcmp($2,"putc") && strcmp($2,"rewind") &&
strcmp($2,"sprint")  else
  {
        insert($2,FUNCTION);
        insert($2,$1);
        for(j=0;j<=k;j++)
        {insertp($2,plist[j]);}
                                k=-1;
  }
  }
  ;
```

```
parameter_list : parameter_list ',' parameter
        | parameter
        ;

parameter : Type ID {plist[++k]=$1;insert($2,$1);insertscope($2,i);}
        ;

Type : INT
  | FLOAT
  | VOID
  ;

CompoundStmt : '{' StmtList '}'
  ;

StmtList : StmtList stmt
  |
  ;

stmt : Declaration
  | if
  | ID '(' ')' ';'
  | while
  | dowhile
  | for
  | RETURN consttype ';' {
                        if(!(strspn($2,"0123456789")==strlen($2)))
                                storereturn(ct,FLOAT);
                        else
                                storereturn(ct,INT); ct++;
                        }
  | RETURN ';' {storereturn(ct,VOID); ct++;}
  | RETURN ID ';' {
        int sct=returnscope($2,stack[top-1]);   //stack[top-1] - current scope
                int type=returntype($2,sct);
        if (type==259) storereturn(ct,FLOAT);
        else storereturn(ct,INT);
        ct++;
        }
  | ';'
  | PRINT '(' STRING ')' ';'
  | CompoundStmt
```

```
  ;

dowhile : DO {dowhile1();} CompoundStmt WHILE '(' E ')' {dowhile2();} ';'
  ;

for   : FOR '(' E {for1();} ';' E {for2();}';' E {for3();} ')' CompoundStmt {for4();}
  ;

if :    IF '(' E ')' {if1();} CompoundStmt {if2();} else
  ;

else : ELSE CompoundStmt {if3();}
  |
  ;

while : WHILE {while1();}'(' E ')' {while2();} CompoundStmt {while3();}
  ;

assignment : ID '=' consttype
  | ID '+' assignment
  | ID ',' assignment
  | consttype ',' assignment
  | ID
  | consttype
  ;

assignment1 : ID {push($1);} '=' {strcpy(st1[++top],"=");} E {codegen_assign();}
  {
        int sct=returnscope($1,stack[index1-1]);
        int type=returntype($1,sct);
        if((!(strspn($5,"0123456789")==strlen($5))) && type==258 && fl==0)
              printf("\nError : Type Mismatch : Line %d\n",printline());
        if(!lookup($1))
        {
              int currscope=stack[index1-1];
              int scope=returnscope($1,currscope);
              if((scope<=currscope && end[scope]==0) && !(scope==0))
              {
                    check_scope_update($1,$5,currscope);
              }
        }
        }
```

```
  | ID ',' assignment1     {
                                    if(lookup($1))
                                        printf("\nUndeclared Variable %s : Line %d\n",$1,printline());
                                    }
  | consttype ',' assignment1
  | ID {
        if(lookup($1))
            printf("\nUndeclared Variable %s : Line %d\n",$1,printline());
        }
//   | function_call
  ;


consttype : NUM
  | REAL
  ;

Declaration : Type ID {push($2);} '=' {strcpy(st1[++top],"=");} E {codegen_assign();} ';'
  {
        if( (!(strspn($6,"0123456789")==strlen($6))) && $1==258 && (fl==0))
        {
            printf("\nError : Type Mismatch : Line %d\n",printline());
            fl=1;
        }
        if(!lookup($2))
        {
            int currscope=stack[index1-1];
            int previous_scope=returnscope($2,currscope);
            if(currscope==previous_scope)
                printf("\nError : Redeclaration of %s : Line %d\n",$2,printline());
            else
            {
                insert_dup($2,$1,currscope);
                check_scope_update($2,$6,stack[index1-1]);
                int sg=returnscope($2,stack[index1-1]);
                g_addr+=4;
            }
        }
        else
        {
            int scope=stack[index1-1];
```

```
                insert($2,$1);
                insertscope($2,scope);
                check_scope_update($2,$6,stack[index1-1]);
                g_addr+=4;
        }
  }

  | assignment1 ';' {
                        if(!lookup($1))
                        {
                                int  currscope=stack[index1-1];
                                int scope=returnscope($1,currscope);
                                if(!(scope<=currscope && end[scope]==0) || scope==0)
                                        printf("\nError : Variable %s out of scope : Line
%d\n",$1,printline());
                        }
                        else
                                printf("\nError : Undeclared Variable %s : Line %d\n",$1,printline());
                        }
/*
  | Type ID '[' assignment ']' ';' {
                                        insert($2,ARRAY);
                                        insert($2,$1);
                                        g_addr+=4;
                                }
*/
        | Type ID '[' assignment ']' ';' {
                int itype;
                if(!(strspn($4,"0123456789")==strlen($4))) { itype=259; } else itype = 258;
                if(itype!=258)
                { printf("\nError : Array index must be of type int : Line %d\n",printline());errc++;}
                if(atoi($4)<=0)
                { printf("\nError : Array index must be of type int > 0 : Line %d\n",printline());errc++;}
                if(!lookup($2))
                {
                        int currscope=stack[top-1];
                        int previous_scope=returnscope($2,currscope);
                        if(currscope==previous_scope)
                        {printf("\nError : Redeclaration of %s : Line %d\n",$2,printline());errc++;}
                        else
                        {
                                insert_dup($2,ARRAY,currscope);
```

```
                        insert_by_scope($2,$1,currscope);     //to insert type to the correct identifier in
case of multiple entries of the identifier by using scope
                                if (itype==258) {insert_index($2,$4);}
                        }
                }
                else
                {
                        int scope=stack[top-1];
                        insert($2,ARRAY);
                        insert($2,$1);
                        insertscope($2,scope);
                        if (itype==258) {insert_index($2,$4);}
                }
        }

  | ID '[' assignment1 ']' ';'
  | STRUCT ID '{' Declaration '}' ';' {
                                        insert($2,STRUCT);
                                        g_addr+=4;
                                        }
  | STRUCT ID ID ';' {
                        insert($3,STRUCT_VAR);
                        g_addr+=4;
                        }
  | error
  ;

array : ID {push($1);}'[' E ']'
  ;

E : E '+'{strcpy(st1[++top],"+");} T{codegen();}
  | E '-'{strcpy(st1[++top],"-");} T{codegen();}
  | T
  | ID {push($1);} LE {strcpy(st1[++top],"<=");} E {codegen();}
  | ID {push($1);} GE {strcpy(st1[++top],">=");} E {codegen();}
  | ID {push($1);} EQ{strcpy(st1[++top],"==");} E {codegen();}
  | ID {push($1);} NEQ{strcpy(st1[++top],"!=");} E {codegen();}
  | ID {push($1);} AND {strcpy(st1[++top],"&&");} E {codegen();}
  | ID {push($1);} OR {strcpy(st1[++top],"||");} E {codegen();}
  | ID {push($1);} '<' {strcpy(st1[++top],"<");} E {codegen();}
  | ID {push($1);} '>' {strcpy(st1[++top],">");} E {codegen();}
  | ID {push($1);} '=' {strcpy(st1[++top],"=");} E {codegen_assign();}
```

```
  | array {array1();}
  ;
T : T '*'{strcpy(st1[++top],"*");} F{codegen();}
  | T '/'{strcpy(st1[++top],"/");} F{codegen();}
  | F
  ;
F : '(' E ')' {$$=$2;}
  | '-'{strcpy(st1[++top],"-");} F{codegen_umin();} %prec UMINUS
  | ID {push($1);fl=1;}
  | consttype {push($1);}
  ;

%%

#include "lex.yy.c"
#include<ctype.h>


int main(int argc, char *argv[])
{
  yon =fopen(argv[1],"r");
  yyparse();
  if(!yyparse())
  {
        printf("Parsing done\n");
        print();
  }
  else
  {
        printf("Error\n");
  }
  fclose(yyin);
  return 0;
}

void yyerror(char *s)
{
  printf("\nLine %d : %s %s\n",yylineno,s,yytext);
}
int printline()
{
  return yylineno;
```

```
}
```

****

Symbol Table file (symbolTable.c)

****

```
#include<stdio.h>
#include<string.h>
struct sym
{
   int sno;
   char token[100];
   int type[100];
   int paratype[100];
   int tn;
   int pn;
   float fvalue;
   int index;
   int scope;
   //int addr;
}st[100];
int n=0,arr[10];
float t[100];
int iter=0;
int returntype_func(int ct)
{
   return arr[ct-1];
}
void storereturn( int ct, int returntype )
{
   arr[ct] = returntype;
   return;
}
void insertscope(char *a,int s)
{
   int i;
   for(i=0;i<n;i++)
   {
          if(!strcmp(a,st[i].token))
          {
                 st[i].scope=s;
                 break;
          }
```

```
  }
}
int returnscope(char *a,int cs)
{
  int i;
  int max = 0;
  for(i=0;i<=n;i++)
  {
        if(!(strcmp(a,st[i].token)) && cs>=st[i].scope)
        {
              if(st[i].scope>=max)
                    max = st[i].scope;
        }
  }
  return max;
}
int lookup(char *a)
{
  int i;
  for(i=0;i<n;i++)
  {
        if( !strcmp( a, st[i].token) )
              return 0;
  }
  return 1;
}
int returntype(char *a,int sct)
{
  int i;
  for(i=0;i<=n;i++)
  {
        if(!strcmp(a,st[i].token) && st[i].scope==sct)
        {
              return st[i].type[0];
        }
  }
}

int returntypearray(char *a)
{
  int i;
  for(i=0;i<=n;i++)
```

```
  {
        if(!strcmp(a,st[i].token) && st[i].tn>1)
        {
              return st[i].type[1];
        }
  }
}

int returntypef(char *a)
{
  int i;
  for(i=0;i<n;i++)
  {
        if(!strcmp(a,st[i].token))
              { return st[i].type[1];}
  }
}

int returntype2(char *a,int sct)
{
  int i;
  for(i=0;i<n;i++)
  {
        if(!strcmp(a,st[i].token) && st[i].scope==sct)
              { return st[i].type[1];}
  }
}

void check_scope_update(char *a,char *b,int sc)
{
  int i,j,k;
  int max=0;
  for(i=0;i<=n;i++)
  {
        if(!strcmp(a,st[i].token)   && sc>=st[i].scope)
        {
              if(st[i].scope>=max)
                    max=st[i].scope;
        }
  }
  for(i=0;i<=n;i++)
  {
```

```
        if(!strcmp(a,st[i].token)   && max==st[i].scope)
        {
                float temp=atof(b);
                for(k=0;k<st[i].tn;k++)
                {
                        if(st[i].type[k]==258)
                                st[i].fvalue=(int)temp;
                        else
                                st[i].fvalue=temp;
                }
        }
  }
}
void storevalue(char *a,char *b,int s_c)
{
  int i;
  for(i=0;i<=n;i++)
  {
        if(!strcmp(a,st[i].token) && s_c==st[i].scope)
        {
                st[i].fvalue=atof(b);
        }
  }
}

void insert(char *name, int type)
{
  int i;
  if(lookup(name))
  {
        strcpy(st[n].token,name);
        st[n].tn=1;
        st[n].type[st[n].tn-1]=type;
        //st[n].addr=address;
        st[n].sno=n+1;
        n++;
  }
  else
  {
        for(i=0;i<n;i++)
        {
                if(!strcmp(name,st[i].token))
```

```
                {
                        st[i].tn++;
                        st[i].type[st[i].tn-1]=type;
                        break;
                }
        }
  }
  return;
}

void insertp(char *name,int type)
{
        int i;
        for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token))
        {
                        st[i].pn++;
                        st[i].paratype[st[i].pn-1]=type;
                        break;
        }
        }
}

void insert_index(char *name,int ind)
{
        int i;
        for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token)  && st[i].type[0]==273)
        {
                        st[i].index = atoi(ind);
        }
  }
}

void insert_by_scope(char *name, int type, int s_c)
{
        int i;
  for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token) && st[i].scope==s_c)
```

```
        {
                        st[i].tn++;
                        st[i].type[st[i].tn-1]=type;
        }
        }
}

int checkp(char *name,int flist,int c)
{
        int i,j;
        for(i=0;i<n;i++)
        {
        if(!strcmp(name,st[i].token))
        {
                        if(st[i].paratype[c]!=flist)
                        return 1;
        }
        }
        return 0;
}

void insert_dup(char *name, int type, int s_c)
{
  strcpy(st[n].token,name);
  st[n].tn=1;
  st[n].type[st[n].tn-1]=type;
  //st[n].addr=addr;
  st[n].sno=n+1;
  st[n].scope=s_c;
  n++;
  return;
}

void print()
{
  int i,j;
  printf("\n.....Symbol Table.....\n");
  printf("..........\n");
  printf("\nSNo.\tToken\t\tValue\t\tScope\t\tType\n");
  printf("..........\n");
  for(i=0;i<n;i++)
  {
```

```
        if(st[i].type[0]==258)
                printf("%d\t%s\t\t%d\t\t%d\t",st[i].sno,st[i].token,(int)st[i].fvalue,st[i].scope);
        else
                printf("%d\t%s\t\t%.1f\t\t%d\t",st[i].sno,st[i].token,st[i].fvalue,st[i].scope);
                printf("\t");
        for(j=0;j<st[i].tn;j++)
        {
                if(st[i].type[j]==258)
                        printf("INT");
                else  if(st[i].type[j]==259)
                        printf("FLOAT");
                else  if(st[i].type[j]==271)
                        printf("FUNCTION");
                else if(st[i].type[j]==269)
                        printf("ARRAY");
                else  if(st[i].type[j]==260)
                        printf("VOID");
                if(st[i].tn>1 && j<(st[i].tn-1))printf(" - ");
        }
        printf("\n");
  }
  printf("-------------------------------------------------------------------------------\n\n");
  return;
}
*****************************************************************************************************
```

Explanation

FILES

parser.l : Lex file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.

parser.y : Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned. This file also contains the semantic rules defined against every production necessary. Rules for producing three address code is also present against the productions and in functions.

symbolTable.c : It is the C file which generates the symbol table. It is included along with the yacc file.

test.c : The input C code which will be parsed and checked for semantic correctness by executing the lex and yacc files along with it.

Yacc file has productions to check the following functionalities:

- Preprocessor directives
- Function definition
- Compound statements
- Nested compound statements
- if else
- Nested while
- Variable definition and declaration
- Assignment operation
- Arithmetic operations

TEST CASES

1. test1.c (arithmetic expressions)

```
#include <stdio.h>
int main()
{
  int a=5;
  int b=6;
  int c=3;
  int d=a+b-c*a/d;
  return 0;
}
```

Output:

```
a = 5

b = 6

c = 3

t0 = a + b

t1 = c * a

t2 = t1 / d

t3 = t0 - t2

d = t3
Parsing done

------------------------------Symbol Table------------------------------
------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
------------------------------------------------------------------------
1       a               5               1               INT
2       b               6               1               INT
3       c               3               1               INT
4       d               0               1               INT
5       main            0.0             0               FUNCTION - INT
------------------------------------------------------------------------
```

2. test2.c (if-else)

```
#include <stdio.h>
int main()
{
  int a=5;
  int b=6;
  if(a<=7)
  {       b=b-4; }
  else
  {       b=b+3; }
  return 0;
}
```

Output:

```
a = 5

b = 6

t0 = a <= 7

t1 = not t0
if t1 goto L1

t2 = b - 4

b = t2

goto L2
L1:

t3 = b + 3

b = t3

L2:
Parsing done

------------------------------Symbol Table------------------------------
------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
------------------------------------------------------------------------
1       a               5               1               INT
2       b               0               1               INT
3       main            0.0             0               FUNCTION - INT
------------------------------------------------------------------------
```

3. test3.c (while)

```
#include <stdio.h>
int main()
{
  int a=5;
  int b=6;
  while(a<20)
  {
        b=b+1;
        a=a+1;
  }
  return 0;
}
```

Output:

```
a = 5

b = 6

L1:

t0 = a < 20

t1 = not t0
if t1 goto L2

t2 = b + 1

b = t2

t3 = a + 1

a = t3

goto L1
L2:
Parsing done

------------------------------Symbol Table------------------------------
------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
------------------------------------------------------------------------
1       a               0               1               INT
2       b               0               1               INT
3       main            0.0             0               FUNCTION - INT
------------------------------------------------------------------------
```

4. test4.c (for loop)

```
#include <stdio.h>
int main()
{
   int a=5;
   int b=6;
   for(a=9;a!=6;a=a-1)
   {
         b=4;
   }
   b=b/9;
   return 0;
}
```

Output:

```
a = 5

b = 6

a = 9

L1:

t0 = a != 6

t1 = not t0
if t1 goto L2
goto L3
L4:

t2 = a - 1

a = t2

goto L1
L3:

b = 4

goto L4
L2:

t3 = b / 9

b = t3
Parsing done

--------------------------------Symbol Table------------------------------
--------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
--------------------------------------------------------------------------
1       a               5               1               INT
2       b               0               1               INT
3       main            0.0             0               FUNCTION - INT
--------------------------------------------------------------------------
```

5. test5.c (do-while)

```
#include <stdio.h>
int main()
{
  int a=5;
  int b=6;
  do
  {
        b=b+1;
  }while(a>7);
  return 0;
}
```

Output:

```
a = 5

b = 6

L1:

t0 = b + 1

b = t0

t1 = a > 7

if t1 goto L1
Parsing done

------------------------------Symbol Table------------------------------
------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
------------------------------------------------------------------------
1       a               5               1               INT
2       b               0               1               INT
3       main            0.0             0               FUNCTION - INT
------------------------------------------------------------------------
```

6. test6.c (nested if-else)

```
#include <stdio.h>
int main()
{
  int a=5; int b=6;
  if(a<=7)
  {
        if(a==9)
        {       b=9;    }
        else
        {       a=10; }
  }
  else
  {      b=2;    }
  return 0;
}
```

Output:

```
a = 5

b = 6

t0 = a <= 7

t1 = not t0
if t1 goto L1

t2 = a == 9

t3 = not t2
if t3 goto L2

b = 9

goto L3
L2:

a = 10

L3:

goto L4
L1:

b = 2

L4:
Parsing done

------------------------------Symbol Table------------------------------
------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
------------------------------------------------------------------------
1       a               10              1               INT
2       b               2               1               INT
3       main            0.0             0               FUNCTION - INT
------------------------------------------------------------------------
```

7. test7.c (nested while)

```
#include <stdio.h>
int main()
{
   int a=5;  int b=6;
   while(a>7)
   {       b=6;
           while(b>=5)
           {
                   a=9;
           }
   }
   return 0;
}
```

Output:

```
a = 5

b = 6

L1:

t0 = a > 7

t1 = not t0
if t1 goto L2

b = 6

L3:

t2 = b >= 5

t3 = not t2
if t3 goto L4

a = 9

goto L3
L4:

goto L1
L2:
Parsing done

--------------------------------Symbol Table-------------------------------
---------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
---------------------------------------------------------------------------
1       a               9               1               INT
2       b               6               1               INT
3       main            0.0             0               FUNCTION - INT
---------------------------------------------------------------------------
```

8. test8.c (array)

```
#include <stdio.h>
int main()
{
   int d=4;
   float c[10];
   int a[10];
   int x=a[d];
   float y=c[d];
   return 0;
}
```

Output:

```
d = 4

t0 = d
t1 = 4*t0
t2 = a + t1

x = t2

t3 = d
t4 = 8*t3
t5 = c + t4

y = t5
Parsing done

------------------------------Symbol Table------------------------------
------------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
------------------------------------------------------------------------
1       d               4               1               INT
2       c               0.0             1                - FLOAT
3       a               0.0             1                - INT
4       x               0               1               INT
5       y               0.0             1               FLOAT
6       main            0.0             0               FUNCTION - INT
------------------------------------------------------------------------
```

RESULTS

The lex (parser.l) and yacc (parser.y) codes are compiled and executed by the following terminal commands to parse the given input file (test.c)

```
lex  parser.l
yacc parser.y
gcc y.tab.c -ll -ly
./a.out test.c
```

After parsing, if there are errors then the line numbers of those errors are displayed along
with a ‘parsing failed’ on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors. Also the three address codes along with the temporary variables are also displayed along with the flow of the conditional and iterative statements.

FUTURE WORK

Intermediate code generator generates three address codes for:

- assignments
- Expressions
- arrays
- conditional statements
- iterative statements

In future, we can extend it to support and generate three-address code for pointers, structures and functions

REFERENCES

1. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
2. https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/
3. http://web.cs.wpi.edu/~kal/courses/compilers/
4. https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm