Report on
" Mini C compiler with C construct nested do while loop withswitchstatements"

Submitted in partial fulfillment of the requirements for SemVI CompilerDesign Laboratory Bachelor of Technology in
Computer Science & Engineering

Submitted by:
Navya Hegde:PES2201800017
Shloka Lakka : PES2201800502
Nayana CS Reddy: PES2201800510

Under the guidance of
Dr .Mehala
Associate Professor

PES University, Bengaluru
January – May 2020
DEPARTMENT OF COMPUTER SCIENCE ANDENGINEERINGFACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013) 100ft Ring Road,
Bengaluru – 560 085, Karnataka, India

INTRODUCTION

A compiler is the program which converts the given source code intoatargetassembly code. It has six phases which are as follows:

1. Lexical Analysis: In the first phase of compiler , the source code is takenasinput and the sequence of tokens is produced. In the lexical analysis , theremoval of whitespaces , comments are taken up.

2. Syntax Analysis: In this phase , the tokens which are sent by the lexical analysis phase are used to generate the syntax tree by using the grammar. Additionally, the symbol table is generated for the variables.

3. Semantic Analysis: In this phase , the syntax tree generated frompreviousstage is taken for type checking etc. The output is the syntax tree.

4. Intermediate Code Generation : In this phase , the intermediate codeisgenerated by using the syntax tree. Also , the IC should have loopintheformof label and if statement.

5. Code optimization: In this phase , code is optimized using the methodslikedead code elimination , constant folding etc. The input is intermediatecodeandhence output is optimized code.

6. Target code generation: In this phase , the optimized code is usedtogeneratethe assembly code. Generally the arm instruction is taken in consideration.

For example:
The input source code is :

```
#include <stdio.h>
int main ()
{

  /* local variable definition */
int a = 10;
int b=64;
do
{
        b=0;
        do
        {
                int v=0;
        }
        while(a);
        a = a + 1;
}
while(1);
return 0;
}
```

The output ie target assembly code is :

```
LDR a R0
MOV R0 #10
L1:

L2:

LDR v R1
MOV R1 #0
CMP a #0
BNE L2

LDR t0 R2
ADD R2 R0 #1
MOV R0 R2
CMP #1 #0
BNE L1

STR a R0
STR v R1
```

LEXICALANALYSIS

In computer science, lexical analysis is the process of convertinga sequenceof characters (such as in a computer program or web page) into a sequenceoftokens (strings with an identified "meaning"). A programthat performs lexicalanalysis may be called a lexer, tokenizer, or scanner (though "scanner"isalsoused to refer to the first stage of a lexer). Such a lexer is generally combinedwith a parser, which together analyze the syntax of programming languages, web pages, and so forth. The script written by us is a computer programcalledthe “lex” program, is the one that generates lexical analyzers ("scanners"or"lexers"). Lex reads an input stream specifying the lexical analyzer andoutputssource code implementing the lexer in the C programming language.

The structure of the lex program consists of three sections: {definition section}
%%
{rules section}
%%
{C code section}

The definition section defines macros and imports header files writteninC. Itisalso possible to write any C code here, which will be copied verbatimintothegenerated source file. The rules section associates regular expressionpatternswith C statements. When the lexer sees text in the input matching a givenpattern, it will execute the associated C code. The C code section containsCstatements and functions that are copied verbatim to generated sourcefile. Aftercompilation in lex , the C called lex.yy.c is generated. Lexical analysis onlytakes care of parsing the tokens and identifying their type. The output of thisphase is the stream of tokens as well as the symbol table representingthetokensand their type.

Lex program:

```
%{
#include<stdio.h>
extern FILE *yyin;
%}

%%
"#include<stdio.h>"|"#include<stdlib.h>"|"#include<string.h>"|"#include<math.h
>" {ECHO;printf("\n<TOKEN,PREROCESSOR>\n");}
auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern
|return|union|continue|for|signed|void|do|if|static|while|default|goto|sizeof|vola
```

```
tile|const|float|short {ECHO;printf("\n<TOKEN,KEYWORD>\n");}[{};,()]

 {ECHO;printf("\n<TOKEN,SEPERATOR>\n");} [+-/=*%]

 {ECHO;printf("\n<TOKEN,OPERATOR>\n");}

([a-zA-Z][0-9])+|[a-zA-Z]* {ECHO;
printf("\n<TOKEN,Identifier>\n");}.|\n ;
(\/\/) {;}
"/*"[^\n]+"*/" {;}
(\/\*.*\*\/) {;}


%%

/*call the yywrap function*/
int yywrap()
 {
return 1;
}

int main(int argc,char **argv)
 {
yyin=fopen(argv[1],"r");
yylex();
fclose(yyin);
return 0;
}
```

In this program , the tokens are printed in the form “<token,type”>andtypecan be identifier, keyword etc. The comments(singleline and multiline) areremoved. “ECHO” command is used to print the token. Input is takeninformof command line arguments.

Input1

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
 {
        int i=0;
        int j=12;


                j = 32*12;
                switch(j)
                {
                case 1: i=9;
                case 2:i=8; break;
                default:j=0;
                }

}
```

Input2

```
#include<stdio.h>
int main ()
{

   /* local variable definition */
int a = 10;

do
{
        char b='A';
        do
        {
                int v=0;
                int c;
        }
        while( a );
        a = a + 1;
}
while(1);
return 0;
}
```

SYNTAX ANALYSIS

In computer science, synatx analysis is the process of where the tokensfromlexical analysis are used to generate the syntax tree using the givengrammar. The phases uses three programs

1. Lex program
2. Yacc program
3. C program to implement symbol table

If the input has error , then accordingly yacc reports the error. The symbol tableconsists of four columns that are name , type , value and scope. The casesofdeclared arrays, assigning values to arrays and errorenous inputs are takencare.The scope is calculated by incrementing the scope variable whichwas initialised to -1 and whenever encountering { then increment and } thendecrement.

The structure of the yacc program consists of three sections:

```
Declarations
%%
Rules
%%
Routines
```

Lex program:

```
%option yylineno

%{
	#include<stdio.h>
	#include"y.tab.h"
	#include<math.h>
	int lno=1;
	int tok_count=0;
	char tokval[100];
	char idname[100];
```

```
        char dectype[100];
        int no_of_entries = 0;
        int scope = 0;
        int enc_lno[100] = {0};
%}
%%
"#include"([ ]+)?((<(\\.|[^>])+>)|(\"(\\.|[^"])+\")) {fputs(yytext,
yyout);++tok_count;return HEADER;}
"#define"[ ]+[a-zA-Z_][a-zA-Z_0-9]* {fputs(yytext,
yyout);++tok_count;returnDEFINE;}
"void"|"char"|"short"|"int"|"long"|"float"|"double"|"signed"|"unsigned" {fputs(yytext,
yyout);++tok_count;strcpy(dectype, yytext);return TYPE_CONST;}
"case"

        {fputs(yytext, yyout);++tok_count;return CASE;} "default"

        {fputs(yytext, yyout);++tok_count;return DEFAULT;} "switch"

        {fputs(yytext, yyout);++tok_count;return SWITCH;} "else"

        {fputs(yytext, yyout);++tok_count;enc_lno[scope] = lno;returnELSE;}"do"

        {fputs(yytext, yyout);++tok_count;return DO;} "while"

        {fputs(yytext, yyout);++tok_count;return WHILE;} "continue"

        {fputs(yytext, yyout);++tok_count;return CONTINUE;} "break"

        {fputs(yytext, yyout);++tok_count;return BREAK;} "return"

        {fputs(yytext, yyout);++tok_count;return RETURN;} "||"

        {fputs(yytext, yyout);++tok_count;return OR_CONST;} "&&"

        {fputs(yytext, yyout);++tok_count;return AND_CONST;}
"=="

        {fputs(yytext, yyout);++tok_count;return E_CONST;} "!="

        {fputs(yytext, yyout);++tok_count;return NE_CONST;} "<="
        {fputs(yytext, yyout);++tok_count;return LE_CONST;} ">="

        {fputs(yytext, yyout);++tok_count;return GE_CONST;} "<"

        {fputs(yytext, yyout);++tok_count;return L_CONST;} ">"

        {fputs(yytext, yyout);++tok_count;return G_CONST;} ">>"

        {fputs(yytext, yyout);++tok_count;return RSHIFT_CONST;}
"<<"

        {fputs(yytext, yyout);++tok_count;return LSHIFT_CONST;}
"++"

        {fputs(yytext, yyout);++tok_count;return INC_CONST;} "--"

        {fputs(yytext, yyout);++tok_count;return DEC_CONST;} "{"

        {fputs(yytext, yyout);++tok_count;return OPEN_SCOPE;} "}"

                              {fputs(yytext, yyout);++tok_count;return CLOSE_SCOPE;}

";"|"="|","|"("|")"|"["|"]"|"*"|"+"|"-"|"/"|"?"|":"|"&"|"|"|"^"|"!"|"~"|"%" {fputs(yytext,
                                   yyout);++tok_count;return
yytext[0];}
"*="

        {fputs(yytext, yyout);++tok_count;return MUL_EQ;} "/="

        {fputs(yytext, yyout);++tok_count;return DIV_EQ;} "+="

        {fputs(yytext, yyout);++tok_count;return ADD_EQ;} "%="

        {fputs(yytext, yyout);++tok_count;return PER_EQ;} ">>="

        {fputs(yytext, yyout);++tok_count;return RS_EQ;} "-="

        {fputs(yytext, yyout);++tok_count;return SUB_EQ;} "<<="
```

```
          {fputs(yytext, yyout);++tok_count;return LS_EQ;} "&="

          {fputs(yytext, yyout);++tok_count;return AND_EQ;} "^="

          {fputs(yytext, yyout);++tok_count;return XOR_EQ;} "|="

          {fputs(yytext, yyout);++tok_count;return OR_EQ;} [0-9]+

          {fputs(yytext, yyout);strcpy(tokval, yytext);++tok_count;yylval.val
=atoi(yytext);return INT;}
[0-9]+"."[0-9]+

          {fputs(yytext, yyout);strcpy(tokval, yytext);++tok_count;yylval.fval
=atof(yytext);return FLOAT;}
"'"."'"

          {fputs(yytext, yyout);strcpy(tokval, yytext);++tok_count;yylval.cval
=yytext[1];return CHAR;}
[a-zA-Z_][a-zA-Z0-9_]*

          {fputs(yytext, yyout);strcpy(idname,
yytext);++tok_count;returnid;}\"(\\.|[^\"])*\"

          {fputs(yytext, yyout);strcpy(tokval,
yytext);++tok_count;returnstring;}"//"(\\.|[^\n])*[\n]
                                                                                              ;
[/][*]([^*]|[*]*[^*/])*[*]+[/] ;[ \t]

          {fputs(yytext, yyout);}
\n

          {fprintf(yyout, "\n%d ",++lno);}
[ ]+
                                                                                            ;
[\n]+

          {fprintf(yyout, "\n%d ",++lno);}
[\t]+
	{fprintf(yyout, "\t");}
%%

int yywrap(void)
{
	//printf("Token count is %d\n",tok_count);
   return 1;
}
```

Yacc program:

```
%{
	#include<stdio.h>
	#include<string.h>
	#include<stdlib.h>
	#include "newSymbolTable.h"
	int yylex(void);
	int yyerror(const char *s);
	int success = 1;
	char name[100];
	// All extern variables


	//TO-DO - Store a stack of scope values along with their
enclosinglinenumbers.
	extern int enc_lno[100];
	extern int no_of_entries;
	extern char tokval[100];
	extern char idname[100];
	extern int lno;
	extern int scope;
	int array = 0;
	// Declaration type of variables
	extern char dectype[100];
%}

%token <val> INT
%token <cval> CHAR
%token <fval> FLOAT
```

```
%token string id TYPE_CONST DEFINE OPEN_SCOPE CLOSE_SCOPE%token IF
FOR DO WHILE BREAK SWITCH CONTINUERETURNCASEDEFAULT GOTO
SIZEOF OR_CONST AND_CONST E_CONSTNE_CONST LE_CONST
GE_CONST G_CONST L_CONSTLSHIFT_CONST MUL_EQ DIV_EQ ADD_EQ
PER_EQRS_EQSUB_EQLS_EQ AND_EQ XOR_EQ OR_EQ RSHIFT_CONST
REL_CONSTINC_CONST DEC_CONST ELSE HEADER Initialization

%right '='
%left '+' '-'
%left '*' '/'

%union{
        int val;
        float fval;
        char cval;
}


%start program_unit
%%
program_unit : HEADER program_unit
                                                     | DEFINE primary_expression
program_unit
                                                          | translation_unit

                                                          ;
translation_unit : external_decl

                                                          | translation_unit external_decl

                                                          ;
                         external_decl : function_definition | decl
                                                          | compound_stat
                                                          ;
function_definition : type_spec declarator compound_stat | declarator compound_stat
                                                          ;
decl : type_spec init_declarator_list ';'

                                                          ;
                                type_spec : TYPE_CONST ;
```

```
init_declarator_list : init_declarator
                                             | init_declarator_list ',' init_declarator
                                                           ;
init_declarator : declarator
                                                                      {

                                                                  strcpy(tokval, "&*&*^&^");
                                                     if(strcmp(dectype, "int")==0
&& scope==0){
                                                                        strcpy(tokval, "0");
                                                                            }
                                                                int k = search((char*)name,
dectype, scope, enc_lno[scope], 1);
                                                                     if(k==-1)
                                                                         s[no_of_entries++] =
init_node((char*)name,scope,lno,dectype, enc_lno[scope], tokval); else if(k==-2){
                                                                    printf("Error at line
number : %d - REDECLARATION\n", lno);
                                                          printf("Aborting...\n");
                                                                        exit(0);
                                                                            }
                                                                      }
                                                              | declarator '=' initializer
                                                                      {
                                                                  char val[100];
                                                                  sprintf(val, "%d", $<val>3);
                                                    //printf("\nInitialization%s
=%s\n\n", name, val);
                                                                int k = search((char*)name,
dectype, scope, enc_lno[scope], 1);
                                                                     if(k==-1)
                                                                         s[no_of_entries++] =
init_node((char*)name,scope,lno,dectype, enc_lno[scope], val); else if(k==-2){
                                                                    printf("Error at line
number : %d - REINITIALIZATION\n", lno);
                                                                        printf("Aborting...\n");
                                                                        exit(0);
                                                                            }
                                                                          else
                                                                        update(k, tokval);
```

```
                                                            }
                                                    ;
spec_qualifier_list : type_spec spec_qualifier_list | type_spec
                                                    ;
             declarator : id {strcpy(name, idname);} | '('declarator')'

                                                 | declarator'['const_expression']'
                                                            {
                                                        //char num[100];
                                                        //sprintf(num, "%d", $3);
                                                        strcat(dectype, (char*)"[");
                                                        //strcat(dectype, num);
                                                        strcat(dectype, (char*)"]");
                                                            }
                                                    | declarator '[' ']'
                                                            {
                                              strcat(dectype, (char*)"[]");
                                                            }
                                                   | declarator '(' param_type_list ')'
                                                            {
                                                        strcpy(tokval, "-");

                                                         enc_lno[scope] =lno;
                                                         s[no_of_entries++] =
     init_node((char*)name,scope,lno,"FUNC", enc_lno[scope], tokval); }
                                                 | declarator '(' func_call_params')'

                                                    | declarator '(' ')'
                                                            {
                                                        strcpy(tokval, "-");

                                                         enc_lno[scope] =lno;
                                                         s[no_of_entries++] =
     init_node((char*)name,scope,lno,"FUNC", enc_lno[scope], tokval); }

                                                    ;
param_type_list : param_list
                                                    ;
param_list : param_decl
                                                    | param_list ',' param_decl
```

```
                                                                                ;
param_decl : type_spec declarator | type_spec abstract_declarator
                                                                                          | type_spec
                                                                                ;
func_call_params : id
                                                                                             | func_call_params ',' id
                                                                            | func_call_params ',' string
                                                                                             | string
                                                                                ;
initializer : assignment_expression {$<val>$=$<val>1;}
                                                                                  | OPEN_SCOPE initializer_list
CLOSE_SCOPE
                                                                               | OPEN_SCOPE initializer_list ','
CLOSE_SCOPE
                                                                ;
initializer_list : initializer
                                                                                   | initializer_list ',' initializer
                                                                                ;
type_name : spec_qualifier_list abstract_declarator| spec_qualifier_list
                                                                                ;
                    abstract_declarator : direct_abstract_declarator ;
direct_abstract_declarator : '(' abstract_declarator ')' | direct_abstract_declarator '['
const_expression ']'
                                                                                          | '[' const_expression ']'
                                                                             | direct_abstract_declarator '[' ']'
                                                                          | '[' ']'
                                                                                | direct_abstract_declarator '('
param_type_list ')'
                                                                             | '(' param_type_list ')'
                                                                             | direct_abstract_declarator '(' ')'
                                                                          | '(' ')'
                                                                                ;
stat : labeled_stat

                                                                                | exp_stat

                                                                                | compound_stat
                                                                                | selection_stat

                                                                                | iteration_stat
```

```
                                                    | jump_stat
                                                    ;
labeled_stat : CASE const_expression ':' stat | DEFAULT ':' stat
                                                    ;
                  exp_stat : expression ';' | ';'
                                                    ;
compound_stat : OPEN_SCOPE lists CLOSE_SCOPE| OPEN_SCOPE
                                         CLOSE_SCOPE
                                                    ;
lists : decl lists
                                                        | stat lists
                                           | decl
                                                    | stat
                                                    ;
   selection_stat : SWITCH {enc_lno[scope] =lno;scope++;}'(' expression ')'
                              compound_stat{scope--;} ;
iteration_stat : DO {scope++;}stat{scope--;} WHILE'(' expression ')' ';' stat
                                                    ;
jump_stat : CONTINUE ';' | BREAK ';'
                                                          | RETURN expression ';'
                                                                  | RETURN ';'
                                                    ;
expression : assignment_expression {$<val>$=$<val>1;}
                                         | expression ',' assignment_expression
                                                    ;
assignment_expression : conditional_expression {$<val>$ =$<val>1;}|
                                         unary_expression '='
assignment_expression
                                                           {
                                              //printf("\nInitialization%s
=%d\n\n", name, $<val>3);
                                                        int k = search((char*)name,
dectype, scope, enc_lno[scope], 0);
                                                         char val[100];
                                                         sprintf(val, "%d", $<val>3);
                                                           if(k==-1)
                                                              s[no_of_entries++] =
init_node((char*)name,scope,lno,dectype, enc_lno[scope],val); else if(k==-2){
                                                          printf("Error at line
number : %d\n", lno);
                                                                                      printf("Aborting...\n");
                                                                                      exit(0);
                                                                                           }
                                                                                        else
                                                                                      update(k, val);
                                                                               //printf("hereeee");

                                                                                    }
                                                      ;
conditional_expression : logical_or_expression {$<val>$ =$<val>1;}|
                                                      logical_or_expression '?' expression
':' conditional_expression
                                                                                    {
                                                                                          if($<val>1){
                                                                                      $<val>$ =$<val>3;
                                                                                          }
                                                                                          else{
                                                                                      $<val>$ =$<val>5;
                                                                                          }
                                                                                    }
                                                                           ;
              const_expression : conditional_expression {$<val>$ =$<val>1;};
logical_or_expression : logical_and_expression {$<val>$ =$<val>1;}|
                                                      logical_or_expression OR_CONST
logical_and_expression
                                                                                    {
                                                                           $<val>$ = $<val>1|| $<val>3;
                                                                                    }
                                                                           ;
          logical_and_expression : inclusive_or_expression {$<val>$ =$<val>1;}|
                                           logical_and_expression
AND_CONST inclusive_or_expression
                                                                                    {
                                                                        $<val>$ = $<val>1&&$<val>3;
                                                                                    }
                                                                           ;
inclusive_or_expression : exclusive_or_expression {$<val>$=$<val>1;}
                                                                      | inclusive_or_expression'|'
exclusive_or_expression
```

```
                                                                                          {
                                                                              $<val>$ = $<val>1| $<val>3;
                                                                                          }
                                                                              ;
exclusive_or_expression : and_expression {$<val>$ =$<val>1;}|
                                                  exclusive_or_expression'^'
and_expression
                                                                                          {
                                                                              $<val>$ = $<val>1^$<val>3;
                                                  }
                                                                              ;
and_expression : equality_expression {$<val>$=$<val>1;}| and_expression '&'
equality_expression
                                                                                          {
                                                                             $<val>$ = $<val>1&$<val>3;
                                                                                          }
                                                                              ;
equality_expression : relational_expression {$<val>$=$<val>1;}|
                                                  equality_expression E_CONST
relational_expression
                                                                                          {
                                                                          $<val>$ = ($<val>1==$<val>3);
                                                                                          }
                                                                           | equality_expression LE_CONST
relational_expression
                                                                                          {
                                                                          $<val>$ = ($<val>1<=$<val>3);
                                                                                          }
                                                                           | equality_expression GE_CONST
relational_expression
                                                                                          {
                                                                          $<val>$ = ($<val>1>=$<val>3);
                                                                                          }
                                                                           | equality_expression NE_CONST
relational_expression
                                                                                          {
                                                                           $<val>$ = ($<val>1!=$<val>3);
                                                                                          }
                                                                              ;
relational_expression : shift_expression {$<val>$ = $<val>1;}| relational_expression
```

```
                                        L_CONST
shift_expression
                                                                {
                                                  $<val>$ = ($<val>1<$<val>3);
                                                                }
                                                  | relational_expression G_CONST
shift_expression
                                                                {
                                                  $<val>$ = ($<val>1>$<val>3);
                                                                }
                                        ;
shift_expression : additive_expression {$<val>$ =$<val>1;}| shift_expression
                                        LSHIFT_CONST
additive_expression
                                                                {
                                                         $<val>$ = ($<val>1<<
$<val>3);
                                                                }
                                             | shift_expression RSHIFT_CONST
additive_expression
                                                                {
                                                         $<val>$ = ($<val>1>>
$<val>3);
                                                                }
                                                   ;
additive_expression : mult_expression {$<val>$ =$<val>1;}| additive_expression '+'
mult_expression
                                                                {
                                                   $<val>$ = $<val>1+$<val>3;
                                                                }
                                                          | additive_expression '-'
mult_expression
                                                                {
                                                   $<val>$ = $<val>1- $<val>3;
                                                                }
                                                   ;
mult_expression : cast_expression {$<val>$ =$<val>1;}| mult_expression '*'
                                        cast_expression
                                                                {
                                                   $<val>$ = $<val>1*$<val>3;
```

```
                                                                        }
                                                      | mult_expression '/' cast_expression
                                                                        {
                                                              $<val>$ = $<val>1/ $<val>3;
                                                                        }
                                                    | mult_expression '%' cast_expression
                                                                        {
                                                             $<val>$ = $<val>1%$<val>3;
                                                                        }
                                                              ;
cast_expression : unary_expression {$<val>$ =$<val>1;}| '(' type_name ')'
                                           cast_expression
                                                              ;
unary_expression : postfix_expression {$<val>$ = $<val>1;}| INC_CONST
                                           unary_expression
                                          | unary_operator cast_expression
                                                              ;
unary_operator : '&' | '*' | '+' | '-' | '~' | '!' ;
postfix_expression : primary_expression {$<val>$=$<val>1;}| postfix_expression '['
                                           expression']'
                                                                     | postfix_expression '('
argument_exp_list ')'
                                                                   | postfix_expression '(' ')'
                                                           | postfix_expression INC_CONST
 {$<val>$ = $<val>1+1;}
                                                              ;
primary_expression : id
                                                                        {
                                                                   strcpy(name, idname);
                                                                   strcpy(dectype, "int");
                                                                  int k = search((char*)name,
dectype, scope, enc_lno[scope], 0);
                                                                   if(k==-1 &&strcmp(name,
"printf")!=0){

      printf("*******ERROR*******\n");
                                                                   printf("There's novariable
by the id : %s of the type %s\n", name, dectype);
                                                                         printf("Aborting...\n");
                                                                         exit(0);
```

```
                                                                                                    }
                                                                                      else if(strcmp(name,
"printf")!=0){
                                                                                    //printf("Got valueof
id : %s as %d", s[k].variable_name, atoi(s[k].value_of_variables)); $<val>$ =
atoi(s[k].value_of_variables);
                                                                                                    }

                                                                                            }

                                                                                   | consts {$<val>$ = $<val>1;}
                                                                  | '(' expression ')' {$<val>$=$<val>2;}
                                                                                  ;
argument_exp_list : assignment_expression | argument_exp_list ','
assignment_expression
                                                                                  ;
consts : INT {
                                                                                   //printf("%d - %d \n", $<val>$,
$1);

                                                                                            }
                                                                                                       | CHAR
                                                                                            {
                                                                                             //printf(".cval = %c\n\n",
$<cval>$);

                                                                                            }
                                                                                                    | FLOAT
                                                                                            {
                                                                                   //printf(".fval = %f", $<fval>$);
                                                                                            }
                                                                                  ;
%%
#include "lex.yy.c"
#include<ctype.h>
char st[100][10];
int top=0;
char i_[2]="0";
char temp[2]="t";


int main(int argc, char *argv[])
```

```
{
        yyin = fopen(argv[1], "r");
        if(!yyparse()){
                printf("\nParsing complete\n");
             printTable();
       }
        else
                printf("\nParsing failed\n");


   printf("\n");
   fclose(yyin);
   return 0;
}

int yyerror(const char *msg)
{
        extern int lno;Initialization;
        printf("Parsing Failed\nLine Number: %d %s\n",lno,msg); success = 0;
        return 0;
}
```

newSymbolTable.c

```
#include "newSymbolTable.h"

struct node init_node(char *variable,int local_scope ,int line_no,char *type,
intenclosing_line_no, char* value){
        struct node node1;
        node1.variable_name = (char*)malloc(sizeof(char)*100);
        node1.value_of_variables = (char*)malloc(sizeof(char)*100); node1.type
        = (char*)malloc(sizeof(char)*100);
        strcpy(node1.variable_name, variable);
        node1.scope_of_variable=local_scope;
        node1.line_no =line_no;
        strcpy(node1.type, type);
        node1.enclosing_line_no=enclosing_line_no;
        strcpy(node1.value_of_variables, value);
        return node1;
}
```

```
int search(char *variable, char *type, int local_scope, int enclosing_line_no,
inttype_dec)
{

        int flag1=0,flag2=0;
        int present=0;
        int k=0;
        int local_k=0;
        for (int i=0;i<no_of_entries;i++)
        {
                if(strcmp(variable,s[i].variable_name)==0)
                {
                                        if((local_scope==s[i].scope_of_variable) &&
(enclosing_line_no==s[i].enclosing_line_no))
                        {

                                flag2=1;
                                local_k=i;
                                present = 1;
                        }
                                        else if(local_scope>s[i].scope_of_variable)
                        {
                                present = 1;
                                k = i;
                                        flag1=s[i].scope_of_variable;
                        }
                        else
                                ;
                }
        }
        if(present==0)
                return -1;
        if(type_dec==1){
                if(flag2==1)
                {
                        printf("--------ERROR--------\n");
                        printf("*******Multiple declarations not allowedfor thesame
scope*******\n");
                        printf("*******Variable %s(%s) declared earlier as
%s*******\n", variable, type, s[local_k].type);
            return -2;
        }
        else
        {
            if(strcmp(s[k].type, type)==0)
                return k;
            return -1;
        }
    }
    else{
        if(flag2==1){
            //printf("gdgdfgfgfdghfgh\n");
        // printf("%s - %s\n", s[local_k].type, type);
            if(strcmp(s[local_k].type, type)==0)
                return local_k;
            return -1;
        }
        else{
            //printf("htfhjgmnvbnbfv\n");
        // printf("%s - %s\n", s[k].type, type);
            if(strcmp(s[k].type, type)==0)
                return k;
            return -1;
        }
    }
}

void update(int k, char *value){
    strcpy(s[k].value_of_variables, value);
    return;
}

void printTable(){
    printf("----------------------SYMBOL TABLE----------------------\n\n");
    printf("Id\t\tVal\tType\tScope\n\n");
    for(int i=0;i<no_of_entries;i++){
        printf("%s\t\t%s\t%s\t%d\n", s[i].variable_name,
s[i].value_of_variables, s[i].type, s[i].scope_of_variable); }
}
```

Input1

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
        int i=0;
        int j=12;


                j = 32*12;
                switch(j)
                {
                case 1: i=9;
                case 2:i=8; break;
                default:j=0;
                }

}
```

Input2

```
#include<stdio.h> int main ()
{

  /* local variable definition */ int a = 10;

do
{
        char b='A';
        do
        {
                int v=0;
                int c[10];
        }
        while( a );
        a = a + 1;
}
while(1);
return 0;
}
```

Input3

```
#include <stdio.h> int main ()
{

   /* local variable definition */ int a = 10;

do
{
        char b='A';
        do
         {
                int v=0;
                int c[3]={0,13,2};
        }
        while( a );
        a = a + 1;
}
while(1);
return 0;
}
```

Input4

```
#include <stdio.h>
int main ()
{

   /* local variable definition */
int a = 10;

do
{
        char b='A';
        do
         {
                int v=0;
                int c[3]=[0,13,2];
        }
        while( a );
        a = a + 1;
}
return 0;
}
```

SEMANTIC ANALYSIS

Semantic analysis is the task of ensuring that the declarations and statementsofa program are Semantically correct, i.e that their meaning is clear andconsistentwith the way in which control structures and data types are supposedtobeused.Semantic analysis can compare information in one part of a parse treetothatin another part (e.g compare reference to variable agrees with its declaration, orthat parameters to a function call match the function definition). Implementingthe semantic actions is conceptually simpler in recursive descent parsingbecause they are simply added to the recursive procedures. Some of thefunctions of Semantic analysis are that it maintains and updates the symbol table, check source programs for semantic errors and warnings like typemismatch, global and local scope of a variable, re-definition of variables, usageof undeclared variables.

Lex program:

1. Do-while

```
D [0-9]
L [a-zA-Z_]
H [a-fA-F0-9]
E [Ee][+-]?{D}+
FS (f|F|l|L)
IS (u|U|l|L)*

%{
#include <stdio.h>
#include<string.h>
#include "y.tab.h"

void count();
void comment();
void scomment();
void new_line();
int check_type();

int line_number=1;
%}

%%
"/*" { comment(); }
"//" { scomment(); }

"break" { count(); return(BREAK); } "char" { count(); return(CHAR); }
"continue" { count(); return(CONTINUE); } "double" { count();
return(DOUBLE); } "else" { count(); return(ELSE); } "float" { count();
return(FLOAT); }
"if" { count(); return(IF); } "while" { count(); return(WHILE); } "do" {
count(); return(DO); } "int" { count(); return(INT); } "long" { count();
return(LONG); } "return" { count(); return(RETURN); } "short" {
count(); return(SHORT); } "sizeof" { count(); return(SIZEOF); } "void" {
count(); return(VOID); } "include" {count(); return(INCLUDE);} "h"
{count(); return(H);}

{L}({L}|{D})* { count(); return(check_type()); }

[\'][.][\'] { count(); return(CHAR_CONSTANT); } {D}+{IS}? { count();
return(INT_CONSTANT); } L?'(\\.|[^\\'])+' { count(); return(CONSTANT); }

{D}+{E}{FS}? { count(); return(CONSTANT); } {D}*"."{D}+({E})?{FS}? {
count(); return(FLOAT_CONSTANT); }{D}+"."{D}*({E})?{FS}? { count();
return(FLOAT_CONSTANT); }

L?\"(\\.|[^\\"])*\" { count(); return(STRING_LITERAL); }

"#" {count(); return(HASH);} "+=" { count(); return(ADD_ASSIGN); }
"-=" { count(); return(SUB_ASSIGN); } "*=" { count();
return(MUL_ASSIGN); } "/=" { count(); return(DIV_ASSIGN); } "%=" {
count(); return(MOD_ASSIGN); } "++" { count(); return(INC_OP); } "--" {
count(); return(DEC_OP); } "&&" { count(); return(AND_OP); } "||" {
count(); return(OR_OP); } "<=" { count(); return(LE_OP); } ">=" { count();
return(GE_OP); } "==" { count(); return(EQ_OP); } "!=" { count();
return(NE_OP); } ";" { count(); return(';'); }
```

```
("{"|"<%") { count(); return('{'); } ("}"|"%>") { count();
return('}'); } "," { count(); return(','); }
":" { count(); return(':'); } "=" { count(); return('='); } "(" {
count(); return('('); } ")" { count(); return(')'); } ("["|"<:") {
count(); return('['); } ("]"|":>") { count(); return(']'); } "." {
count(); return('.'); } "&" { count(); return('&'); }
"!" { count(); return('!'); } "~" { count(); return('~'); } "-" {
count(); return('-'); } "+" { count(); return('+'); }
"*" { count(); return('*'); } "/" { count(); return('/'); } "%" {
count(); return('%'); } "<" { count(); return('<'); } ">" { count();
return('>'); } "^" { count(); return('^'); } "|" { count(); return('|');
} "?" { count(); return('?'); }
"\n" { count(); new_line(); } [ \t\v\n\f] { count(); }
. { /* ignore bad characters */ } %%

yywrap()
{
        return(1);
}

void new_line()
{
        line_number++;
}
void comment()
{
        char c, c1;

loop:
        while ((c = input()) != '*' && c != 0) continue;
        if ((c1 = input()) != '/' && c != 0) {
                /* unput(c1); */
                goto loop;
        }

}

void scomment()
{
        char c, c1;
        while((c=input()) !='\n' && c!=0) continue;
                /* putchar(c); */
        if(c!=0)
                putchar('\n');

}

int column = 0;

void count()
{
        strcpy(yylval.string,yytext);
        int i;
        for (i = 0; yytext[i] != '\0'; i++)
                if (yytext[i] == '\n')
                        column = 0;
                else if (yytext[i] == '\t')
                        column += 8 - (column % 8); else
                        column++;
        ECHO;
}


int check_type()
{
/*
* pseudo code --- this is what it should check *
* if (yytext == type_name)
* return(TYPE_NAME);
*
* return(IDENTIFIER);
*/

/*
* it actually will only return IDENTIFIER */
        /* printf("IDENTIFIER"); */
        return(IDENTIFIER);
}
/*
int main()
```

```
{
        yyin = fopen(argv[1], "r");
        yylex();
        fclose(yyin);
} */
```

2. Switch

```
%{
#include<stdio.h>
#include "y.tab.h"
extern YYSTYPE yylval;
int flag=0;
int count = 0;
%}
%option yylineno
%%
"//"[' 'a-zA-Z0-9.]* ;
\/\*(.*\n)*.*\*\/ ;
int {ECHO; yylval.var_type=strdup(yytext);return INT;} float {ECHO;
yylval.var_type=strdup(yytext);return FLOAT;} char {ECHO;
yylval.var_type=strdup(yytext);return CHAR;} "case" { ECHO; return CASE;}
switch {ECHO;return SWITCH;}
default {ECHO;return DEFAULT;}
"'" {ECHO;return SINGLE;}
"," {ECHO;return COMMA;}
":" {ECHO;return COLON;}
";" {ECHO; return SC;}
[0-9]+((\.[0-9]+)?) {ECHO;yylval.text=strdup(yytext);return NUM;}
"." {ECHO; return DOT;}


"+=" {ECHO; yylval.text=strdup(yytext); return SPLUS;} "-=" {ECHO;
yylval.text=strdup(yytext); return SMINUS;} "*=" {ECHO;
yylval.text=strdup(yytext); return SMULT;} "/=" {ECHO;
yylval.text=strdup(yytext); return SDIV;} "++" {ECHO;
yylval.text=strdup(yytext); return INC;} "--" {ECHO; yylval.text=strdup(yytext);
return DEC;} "(" {ECHO;return OPEN;}
")" {ECHO;return CLOSE;}
"<=" {ECHO;yylval.text=strdup(yytext); return LESEQ; } ">="
{ECHO;yylval.text=strdup(yytext); return GRTEQ;} "!=" {ECHO;
yylval.text=strdup(yytext); return NOTEQ;} "==" {ECHO;
yylval.text=strdup(yytext); return EQEQ; } "<" {ECHO;
yylval.text=strdup(yytext); return LESS;}
">" {ECHO; yylval.text=strdup(yytext); return GREAT;} "{" {ECHO; return
OPBRACE;}
"}" {ECHO;return CLBRACE;}
"+" {ECHO;yylval.text=strdup(yytext); return PLUS;} "-"
{ECHO;yylval.text=strdup(yytext); return MINUS;} "="
{ECHO;yylval.text=strdup(yytext); return ASSIGN; } "*"
{ECHO;yylval.text=strdup(yytext); return MULT;} "/"
{ECHO;yylval.text=strdup(yytext); return DIV;} "^"
{ECHO;yylval.text=strdup(yytext); return POW;} "||"
{ECHO;yylval.text=strdup(yytext); return OR;} "&&"
{ECHO;yylval.text=strdup(yytext); return AND;} [' '\n\t\s] {ECHO;}
continue {ECHO;return CONTINUE;} break {ECHO;return
BREAK;}
return {ECHO;return RETURN;}
if {ECHO;return IF;}
while {ECHO;return WHILE;}
for {ECHO;return FOR;}
else {ECHO;return ELSE;}
main {ECHO; return MAIN;}

[a-zA-Z]+[a-zA-Z0-9]* {ECHO;yylval.text=strdup(yytext);
return(ID);}# return 0;
%%
```

Yacc program:

1. Switch

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <stdarg.h>
void yyerror(const char*);
int yylex();
int scope[100];
int scope_ctr;
int scope_ind;
char typ[10]="nothing";
typedef struct AST{
        char lexeme[100];
        int NumChild;
        struct AST **child;
}AST_node;

struct AST* make_for_node(char* root, AST_node* child1,
AST_node*child2,AST_node* child3, AST_node* child4);
struct AST * make_node(char*, AST_node*, AST_node*);  struct
AST* make_leaf(char* root);
void AST_print(struct AST *t);
extern FILE* yyin;
extern int yylineno;
%}

%token DOT SINGLE SC COMMA LETTER OPBRACE CLBRACECONTINUE
BREAK IF ELSE FOR WHILE POWOPENCLOSECOMMENT

%union {char* var_type; char* text; struct AST *node;}

%token <var_type> INT FLOAT CHAR %token <text> ID NUM PLUS MINUS
MULT DIV ANDORLESSGREATLESEQ GRTEQ NOTEQ EQEQ ASSIGN  SPLUS
SMINUS SMULTSDIVINC DEC SWITCH
%token <node> MAIN RETURN DEFAULT CASE COLON

%type <var_type> Type
%type <text> Varlist relOp logOp s_op
%type <node> F T E assign_expr1 assign_expr relexp logexp conddecl unary_expr
iter_stat stat comp_stat start jump_stat select_stat STCBDs_operation


%%

start:INT MAIN OPEN CLOSE comp_stat {$$ = make_leaf($1);
$$=make_node("Main",$1,$5);printf("\n\nAST:\n\n");
AST_print($$);YYACCEPT;}

    ;

comp_stat: OPBRACE SCOPE stat CLBRACE {$$=$3;} ;
```

```
SCOPE: {scope_ctr++;scope[scope_ind++]=scope_ctr;} ;
stat:E SC stat {$$=make_node("Stat",$1,$3);} |assign_expr stat
   {$$=make_node("Stat",$1,$2);} |comp_stat stat
   {$$=make_node("Stat",$1,$2);} |select_stat stat
   {$$=make_node("Stat",$1,$2);} |iter_stat stat
   {$$=make_node("Stat",$1,$2);} |jump_stat stat
   {$$=make_node("Stat",$1,$2);} |decl stat {$$=make_node("Stat",$1,$2);}
   | {$$=make_leaf(" ");}
   ;


ST : SWITCH OPEN ID CLOSE OPBRACE B CLBRACE
{$3=make_leaf($3);$$=make_node("Switch",$3,$6);} ;


B : C {$$=$1;}
   | C D {$$=make_node("Cases",$1,$2);}
   | C B {$$=make_node("Cases",$1,$2);}
   ;

C : CASE NUM COLON stat {$$=make_node("Case",$2,$4);};
D : DEFAULT COLON stat {$1=make_leaf(" ");
$$=make_node("Default",$1,$3);}
   ;




select_stat: ST {$$=$1;}
                 ;

            iter_stat:FOR OPEN decl cond SC E CLOSE comp_stat
                      {$$=make_for_node("for",$3,$4,$6,$8);}
                 ;

jump_stat:CONTINUE SC {$$=make_leaf("Continue");} | BREAK SC
                 {$$=make_leaf("Break");}
                 |RETURN E SC {$1=make_leaf("Return");$$ = make_node("Stat",$1,$2);}
```

```
                ;

cond:relexp {$$ = $1;}
        |logexp {$$ = $1;}
        |E {$$ = $1;}
        ;


relexp:E relOp E {$$=make_node($2,$1,$3);} ;

logexp:E logOp E {$$=make_node($2,$1,$3);} ;

logOp:AND {$$ = $1;}
         |OR {$$ = $1;}
         ;

relOp:LESEQ {$$ = $1;}
    |GRTEQ {$$ = $1;}
    |NOTEQ {$$ = $1;}
    |EQEQ {$$ = $1;}
         |LESS {$$ = $1;}
         |GREAT {$$ = $1;}
         ;

decl:Type Varlist SC {$1=make_leaf($1); $$=make_leaf($2);
$$=make_node("VarDecl",$1,$2); }
   |Type assign_expr1 {$1=make_leaf($1); $$=make_node("VarDecl",$1,$2);};

Type:INT {$$ = $1;strcpy(typ,$1);} |FLOAT {$$ =
        $1;strcpy(typ,$1);}
        ;

Varlist:Varlist COMMA ID
             {$3=make_leaf($3);$$=make_node("VarList",$1,$3);} |ID
                                {$$=make_leaf($1);}


           assign_expr:ID ASSIGN E COMMA assign_expr {$1=make_leaf($1);
            $$=make_for_node($2,$1,$3,make_leaf(","),$5);} |ID ASSIGN E SC
                                      {$1=make_leaf($1);
```

```
$$=make_node($2,$1,$3);}

                    ;

                assign_expr1:ID ASSIGN E COMMA assign_expr1
  {$1=make_leaf($1);$$=make_for_node($2,$1,$3,make_leaf(","),$5);}|ID
                                        ASSIGN E SC
 {$1=make_leaf($1);$$=make_node($2,$1,$3);} ;

E:E   PLUS   T   {$$=make_node($2,$1,$3);   }   |E   MINUS   T
 {$$=make_node($2,$1,$3);} |T {$$=$1;}
 ;

T:T MULT F {$$=make_node($2,$1,$3);} |T DIV F
 {$$=make_node($2,$1,$3);}
 |F {$$=$1;}
 ;

F:ID {$$=make_leaf($1);}
 |NUM {$$=make_leaf($1);}
 |OPEN E CLOSE {$$=$2;}
 |unary_expr {$$=$1;}
 |s_operation {$$=$1;}
 ;

s_operation: ID s_op ID {$1=make_leaf($1); $3=make_leaf($3);
$$=make_node($2,$1,$3);}
                        | ID s_op NUM {$1=make_leaf($1); $3=make_leaf($3);
$$=make_node($2,$1,$3);}
                                    | ID s_op OPEN E CLOSE {$1=make_leaf($1);
$$=make_node($2,$1,$4);}
                        ;

s_op:SPLUS {$$=$1;}
        |SMINUS {$$=$1;}
        |SMULT {$$=$1;}
        |SDIV {$$=$1;}
        ;

unary_expr:INC ID {$$=make_leaf($1);
```

```
          $$=make_leaf($2);$$=make_node("temp",$1,$2);} |ID INC
                              {$$=make_leaf($1);
         $$=make_leaf($2);$$=make_node("temp",$1,$2);} |DEC ID
                              {$$=make_leaf($1);
         $$=make_leaf($2);$$=make_node("temp",$1,$2);} |ID DEC
                              {$$=make_leaf($1);
       $$=make_leaf($2);$$=make_node("temp",$1,$2);} | MINUS ID
                              {$$=make_leaf($1);
$$=make_leaf($2);$$=make_node("temp",$1,$2);} | MINUS NUM
                {$$=make_leaf($1);
$$=make_leaf($2);$$=make_node("temp",$1,$2);} ;
%%
void yyerror(const char* arg)
{
        printf("%s\n",arg);
}


void AST_print(struct AST *t)
{
        static int ctr=0;
        //printf("inside print tree\n");
        int i;
        if(t->NumChild==0)
                return;

        struct AST *t2=t;
        printf("\n%s -->",t2->lexeme);
        for(i=0;i<t2->NumChild;++i)
        {
                printf("%s ",t2->child[i]->lexeme);
        }
        for(i=0;i<t2->NumChild;++i)
        {
                AST_print(t->child[i]);
        }


}
```

```
struct AST* make_node(char* root, AST_node* child1,
AST_node*child2){
        //printf("Creating new node\n");
        struct AST * node = (struct AST*)malloc(sizeof(struct AST));
        node->child = (struct AST**)malloc(2*sizeof(struct AST*));
        node->NumChild = 2;//
        strcpy(node->lexeme,root);
        //printf("Copied lexeme\n");
        //printf("%s\n",node->lexeme);
        node->child[0] = child1;
        node->child[1] = child2;
        return node;
}

struct AST* make_for_node(char* root, AST_node* child1,
AST_node*child2,AST_node* child3, AST_node* child4)
{
        //printf("Creating new node\n");
        struct AST * node = (struct AST*)malloc(sizeof(struct AST));
        node->child = (struct AST**)malloc(4*sizeof(struct AST*));
        node->NumChild = 4;
        strcpy(node->lexeme,root);
        node->child[0] = child1;
        node->child[1] = child2;
        node->child[2] = child3;
        node->child[3] = child4;
        return node;
}


struct AST* make_leaf(char* root)
{
        //printf("Creating new leaf ");
        struct AST * node = (struct AST*)malloc(sizeof(struct AST));
        strcpy(node->lexeme,root);
        //printf("%s\n",node->lexeme);
        node->NumChild = 0;
        node->child = NULL;
        return node;
}



int main(int argc,char **argv)
{
        yyin=fopen(argv[1],"r");
        if(!yyparse())
        {
                printf("Success\n");

        }
        else
        {
                printf("Fail\n");
        }

        return 0;
}
```

2. Do-while

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define COUNT 10 int curr_scope = 0; int
insideloop = 0; int opening_brackets = 0; int
closing_brackets = 0; int nesting = 0;

   typedef struct Node{ struct Node *left;
        struct Node *right;
        char token[100];
        struct Node *val;
}Node;

      typedef struct tree_stack{ Node *node;
        struct tree_stack *next; }tree_stack;

typedef struct Trunk {
        struct Trunk *prev;
        char str[100];
}Trunk;


void push_scope(int);
void pop_scope();
int peep_scope();
void create_node(char *token, int leaf); void
push_tree(Node *newnode); Node* pop_tree();
Node* pop_tree_2();
void printtree(Node *tree); void printTree(Node*
root, Trunk*,int); void showTrunks(Trunk *p);

extern char yytext[];
extern int line_number;
extern int column;
extern FILE *yyin;
//Global variables
tree_stack *tree_top = NULL;
%}

%union
{
        int ival;
        char string[128];
}

%token IDENTIFIER CONSTANT CHAR_CONSTANT
INT_CONSTANTFLOAT_CONSTANT STRING_LITERAL SIZEOF %token
INC_OP DEC_OP LE_OP GE_OP EQ_OP NE_OP H%token AND_OP OR_OP
MUL_ASSIGN DIV_ASSIGNMOD_ASSIGNADD_ASSIGN SUB_ASSIGN
%token CHAR SHORT INT LONG FLOAT DOUBLE VOID

%token IF ELSE WHILE DO CONTINUE BREAK RETURN%token HASH
INCLUDE LIBRARY

%type <string> IDENTIFIER CONSTANT CHAR_CONSTANTFLOAT_CONSTANT
INT_CONSTANT STRING_LITERALSIZEOFINC_OP DEC_OP LE_OP GE_OP
EQ_OP NE_OP H AND_OPOR_OPMUL_ASSIGN DIV_ASSIGN MOD_ASSIGN
ADD_ASSIGNSUB_ASSIGNCHAR SHORT INT LONG FLOAT DOUBLE VOID
```

```
%type <string> IF ELSE CONTINUE BREAK RETURNHASHINCLUDELIBRARY
%type <string> declaration init_declarator_list init_declarator
type_specifierdeclarator logical_and_expression logical_or_expression
conditional_expression assignment_expression initializer %type <string>
primary_expression postfix_expression
unary_expressionmultiplicative_expression additive_expression
relational_expressionequality_expression expression initializer_list

%start hashinclude
%%

primary_expression
        : IDENTIFIER { strcpy($$, $1); create_node($1, 1);} | CHAR_CONSTANT
    { strcpy($$, $1); create_node($1, 1);}| FLOAT_CONSTANT{ strcpy($$, $1);
    create_node($1, 1);}| CONSTANT { strcpy($$, $1); create_node($1, 1);}
    |    INT_CONSTANT    {    strcpy($$,    $1);    create_node($1,    1);}|
        STRING_LITERAL   {   strcpy($$,   $1);   create_node($1,   1);}|  '('
        expression ')' {}
        ;

postfix_expression
        : primary_expression { strcpy($$, $1); }
        | postfix_expression '[' expression ']' {

        pop_tree();

        pop_tree();
                                                                                                    char
s[30];
                                                                                                    strcpy(s,
$1);
                                                                                                    strcat(s,
"[");
                                                                                                    strcat(s,
$3);
                                                                                                    strcat(s,
"]");

        create_node(s, 1);
        strcpy($$, s);

                                                                                              }
        | postfix_expression '(' ')' {}
        | postfix_expression '(' argument_expression_list ')' {} | postfix_expression '.'
        IDENTIFIER {} | postfix_expression INC_OP {

                create_node("1",1);
                create_node("+", 0);

        }
        | postfix_expression DEC_OP {

                create_node("1",1);
                create_node("-", 0);
        }
        ;

argument_expression_list
        : assignment_expression {}
        | argument_expression_list ',' assignment_expression {} ;

unary_expression
        : postfix_expression { strcpy($$, $1); } | INC_OP
        unary_expression {

                create_node("1",1);
                create_node("+", 0);

        }
        | DEC_OP unary_expression {

                create_node("1",1);
                create_node("-", 0);

        }
        | unary_operator unary_expression {}
        | SIZEOF unary_expression {}
        | SIZEOF '(' type_specifier ')' {}
        ;

unary_operator
        : '&' {}
        | '*' {}
        | '+' {}
        | '-' {}
        | '~' {}
        | '!' {}
        ;

multiplicative_expression
        : unary_expression { strcpy($$, $1); } | multiplicative_expression '*'
        unary_expression {

                create_node("*", 0);
        }
        | multiplicative_expression '/' unary_expression { create_node("/", 0);


        }
        | multiplicative_expression '%' unary_expression { create_node("%", 0);


        }
        ;

additive_expression
        : multiplicative_expression { strcpy($$, $1); } | additive_expression '+'
        multiplicative_expression {

                create_node("+", 0);

        }
        | additive_expression '-' multiplicative_expression { create_node("-", 0);

        }
        ;

relational_expression
```

```
        : additive_expression { strcpy($$, $1); }
        | relational_expression '<' additive_expression { create_node("<", 0);


        }
        | relational_expression '>' additive_expression { create_node(">", 0);

        }
        | relational_expression LE_OP additive_expression { create_node("<=", 0);


        }
        | relational_expression GE_OP additive_expression { create_node(">=", 0);



        }
        ;

equality_expression
        : relational_expression { strcpy($$, $1); } | equality_expression EQ_OP
        relational_expression {

                create_node("==", 0);



        }
        | equality_expression NE_OP relational_expression { create_node("!=", 0);



        }
        ;

logical_and_expression
        : equality_expression { strcpy($$, $1); } | logical_and_expression AND_OP
        equality_expression {

                        create_node("&&", 0);
```

```
            }
        ;

logical_or_expression
        : logical_and_expression { strcpy($$, $1); } | logical_or_expression OR_OP
        logical_and_expression {create_node("||", 0);}
        ;

conditional_expression
        : logical_or_expression{ strcpy($$, $1); } | logical_or_expression
        '?' expression ':' conditional_expression{};

assignment_expression
        : conditional_expression { strcpy($$, $1); } | unary_expression
        assignment_operator assignment_expression{

                                                //addval($1, $3, curr_scope);

                                                create_node("=", 0);

                        }
        ;

assignment_operator
        : '=' {}
        | MUL_ASSIGN {}
        | DIV_ASSIGN {}
        | MOD_ASSIGN {}
        | ADD_ASSIGN {}
        | SUB_ASSIGN {}

        ;

expression
        : assignment_expression { strcpy($$, $1); } | expression ','
        assignment_expression {} ;
parameter_list
        : parameter_declaration {}
        | parameter_list ',' parameter_declaration {}
        ;
```

```
parameter_declaration
        : type_specifier declarator {}//{ lookup($1, $2); }| type_specifier {}
        ;

identifier_list
        : IDENTIFIER {

        }
        | identifier_list ',' IDENTIFIER {}
        ;


initializer
        : assignment_expression { strcpy($$, $1); } | '{' initializer_list '}' {
                                                                                    char s[20] ="{";
                                                                                   strcat(s, $2);
                                                                                   strcat(s, "}");
                                                                                   strcpy($$, s);
                                                                             for(int i=0; s[i]!='}'; i++)
                                                                                                {
                                                                                                 Node *n;
                                                                                               if(s[i]==',')
                                                                                  n=pop_tree();
                                                                                                }
                                                                                   pop_tree();
                                                                                   create_node(s, 1);
                                                                                        }
        | '{' initializer_list ',' '}' {
                                                                                            char s[20] ="{";
                                                                                           strcat(s, $2);
                                                                                           strcat(s, ",}");
                                                                                           strcpy($$, s);
                                                                                     create_node(s, 1);
                                                                                                }
        ;

initializer_list
        : initializer { strcpy($$, $1);}
        | initializer_list ',' initializer {}
        ;

statement //HERE YOU MADE CHANGE : compound_statement
        | expression_statement {}
        | selection_statement {}
        | iteration_statement {}
        | jump_statement {}
    | declaration {}
        ;


compound_statement
        : compound_statement_types
        ;

compound_statement_types
        : '{' '}'
        | '{' statement_list '}'
        ;

declaration
        : type_specifier init_declarator_list ';' { // printf("%s %s", $1, $2);

                                                                            }
        ;

init_declarator_list
        : init_declarator {strcpy($$,$1);}
        | init_declarator_list ',' init_declarator { strcpy($$,$3);

        strcat($$, ",");
        strcat($$, $1);
                                                                                        }
        ;

init_declarator
        : declarator {strcpy($$,$1);}
        | declarator '=' initializer {
                                                                                create_node("=", 0);
                                                                                    char val[20];
```

```
                                                                        strcpy(val, $3);
                                                                                }

        ;

type_specifier
        : VOID {strcpy($$,$1);}
        | CHAR {}
        | SHORT {}
        | INT {strcpy($$,$1);}
        | LONG {}
        | FLOAT {}
        | DOUBLE {}
        ;

declarator
        : IDENTIFIER { create_node($1, 1); strcpy($$,$1);} /* | '(' declarator ')' */
        | declarator '[' INT_CONSTANT ']' {
                                                                           Node *n=
pop_tree();
                                                                        char s[30] ="";
                                                                        strcat(s, $1);
                                                                        strcat(s,"[");
                                                                        strcat(s, $3);
                                                                        strcat(s, "]");
                                                                    create_node(s, 1);
                                                                                }
        | declarator '(' parameter_list ')' {}
        | declarator '(' identifier_list ')' {}
        | declarator '(' ')' {strcpy($$, $1); }
        ;
statement_list
        : statement {}
        | statement_list statement { create_node("stmt", 0);} ;

expression_statement
        : ';' {}
        | expression ';' {}
        ;
```

```
selection_statement
        : IF '(' expression ')' compound_statement { create_node("if", 0);}| IF '('
expression ')' compound_statement ELSE
compound_statement{create_node("else", 0); create_node("if", 0);};

iteration_statement
        : WHILE '(' expression ')' {insideloop = 1; } compound_statement { insideloop
= 0; create_node("while", 0);}
        | DO {insideloop = 1; } compound_statement WHILE'(' expression')' ';'
 {insideloop = 0; create_node("do-while", 0);} ;


jump_statement
        : CONTINUE ';' {}
        | BREAK ';' {
                                                        if(!insideloop)
                                                              {
                                                         printf("\n%*s\n%*s <--- break
statement not within loop \n", column, "^", column);
                                                           exit(0);
                                                              }
                                   }
        | RETURN ';' { create_node("return",1);} | RETURN expression ';' { char
        s[20] = "return ";strcat(s, $2);}

        ;

hashinclude
                              : HASH INCLUDE '<' IDENTIFIER '.' H '>' hashinclude {}
        | translation_unit {} //{display(st);}
        ;

translation_unit
        : external_declaration {}
        | translation_unit external_declaration {} ;

external_declaration
        : function_definition {}
        | declaration {}
        ;

function_definition
        : type_specifier declarator compound_statement {} {}| declarator declaration
        compound_statement {} | declarator compound_statement {}
        ;

%%

void create_node(char *token, int leaf)
{
        Node *l;
        Node *r;
        if(leaf==0)
        {
                r = pop_tree();
                l = pop_tree();
        }
        else if(leaf ==1)
        {
                l = NULL;
                r = NULL;
        }
        else
        {
                l = pop_tree();
                r = NULL;
        }
        Node *newnode = (Node*)malloc(sizeof(Node));
        strcpy(newnode->token, token);
        newnode->left = l;
        newnode->right = r;
        push_tree(newnode);
}
void push_tree(Node *newnode)
{
        tree_stack *temp= (tree_stack*)malloc(sizeof(tree_stack)); temp->node
        = newnode;
        temp->next = tree_top;
        tree_top = temp;
```

```
}
void modify_top(char *s)
{
        strcpy(tree_top->node->token, s);
}

Node* pop_tree()
{
        if(tree_top==NULL)
                return NULL;
        tree_stack *temp = tree_top;
        tree_top = tree_top->next;
        Node *retnode = temp->node;
        free(temp);
        return retnode;
}
Node* pop_tree_2()
{
        if(tree_top==NULL)
                return NULL;
        tree_stack *temp = tree_top->next;
        tree_top->next = tree_top->next->next;
        Node *retnode = temp->node;
        free(temp);
        return retnode;
}
// Helper function to print branches of the binary tree void
showTrunks(Trunk *p)
{
        if (p == NULL)
                return;
        showTrunks(p->prev);

        printf("%s",p->str);
}

// Recursive function to print binary tree // It uses inorder
traversal
void printTree(Node *root, Trunk *prev, int isLeft) {
        if (root == NULL)
                return;

        char prev_str[100] = " ";
        Trunk *trunk = (Trunk*)malloc(sizeof(Trunk)); trunk->prev
        = prev;
        strcpy(trunk->str, prev_str);

        printTree(root->right, trunk, 1);

          if (!prev) //if prev == NULL strcpy(trunk->str,"---");
        else if (isLeft)
        {
                strcpy(trunk->str,".---");
                strcpy(prev_str,"\t |");
        }
        else
        {
                strcpy(trunk->str,"`---");
                strcpy(prev->str,prev_str);
        }

        showTrunks(trunk);
        printf(" %s\n",root->token);
        if (prev)
                strcpy(prev->str,prev_str);
        strcpy(trunk->str,"\t |");

        printTree(root->left, trunk, 0);
}

void printtree(Node *tree)
{
        int i;
        if (tree->left || tree->right)
                printf("(");
        printf(" %s ", tree->token);
        if (tree->left)
                printtree(tree->left);
        if (tree->right)
                printtree(tree->right);
        if (tree->left || tree->right)
                printf(")");
}

yyerror(s)
char *s;
{
        fflush(stdout);

                printf("\n%*s\n%*s\n", column, "^", column, s);


}

int main(int argc,char **argv)
{
        yyin = fopen(argv[1], "r");
        tree_top = (tree_stack*)malloc(sizeof(tree_stack));
        tree_top->node = NULL;
        tree_top->next = NULL;
        struct Node *root;
        yyparse();
        root = pop_tree();

        printtree(root);
        printf("\n");
        printTree(root, NULL, 0);
        fclose(yyin);

}
```

INTERMEDIATE CODE GENERATION

Here , the intermediate code is generated using the syntax tree whichwasobtained from the previous stage . It should be noted that whenever we encounter the program which loops , then it should be dealt by usingif statements and labels.

Source code

1. Do while :

Lex :

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


%%
[ \t] ;
\n {yylineno++;}
"{" {open1(); return '{';}
"}" {close1(); 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;
printfreturn 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:

```
%{
        #include <stdio.h>
        #include <stdlib.h>

        int g_addr = 100;
int i=1,lnum1=0,label1[20],ltop1;
int
stack[100],index1=0,end[100],arr[10],gl1,gl2,ct,c,b,fl,top=0,label[20],lnum=0,l
top=0;
char st1[100][10];
char i_[2]="0";
char temp[2]="t";
char null[2]=" ";
void yyerror(char *s);
int printline();
void open1()
{
        stack[index1]=i;
        i++;
        index1++;
        return;
}
void close1()
{
        index1--;
        end[stack[index1]]=1;
        stack[index1]=0;
        return;
}
void if1()
{
        lnum++;
        strcpy(temp,"t");
        strcat(temp,i_);
        printf("%s = not %s\n",temp,st1[top]);
        printf("if %s goto L%d\n",temp,lnum);
        i_[0]++;
        label[++ltop]=lnum;
}
```

```
void if2()
{
        lnum++;
        printf("goto L%d\n",lnum);
        printf("L%d: \n",label[ltop--]);
        label[++ltop]=lnum;
}
void if3()
{
        printf("L%d:\n",label[ltop--]);
}
void w1()
{
        lnum++;
        label[++ltop]=lnum;
        printf("L%d:\n",lnum);
}
void w2()
{
        lnum++;
        strcpy(temp,"t");
        strcat(temp,i_);
        printf("%s = not %s\n",temp,st1[top--]); printf("if
        %s goto L%d\n",temp,lnum); i_[0]++;
        label[++ltop]=lnum;
}
void w3()
{
        int y=label[ltop--];
        printf("goto L%d\n",label[ltop--]);
        printf("L%d:\n",y);
}
void dw1()
{
        lnum++;
        label[++ltop]=lnum;
        printf("L%d:\n",lnum);
}
void dw2()
{
        printf("if %s goto L%d\n",st1[top--],label[ltop--]); }
void f1()
{
        lnum++;
        label[++ltop]=lnum;
        printf("L%d:\n",lnum);
}
void f2()
{
        lnum++;
        strcpy(temp,"t");
        strcat(temp,i_);
        printf("%s = not %s\n",temp,st1[top--]); printf("if
        %s goto L%d\n",temp,lnum); i_[0]++;
        label[++ltop]=lnum;
        lnum++;
        printf("goto L%d\n",lnum);
        label[++ltop]=lnum;
        lnum++;
        printf("L%d:\n",lnum);
        label[++ltop]=lnum;
}
void f3()
{
        printf("goto L%d\n",label[ltop-3]);
        printf("L%d:\n",label[ltop-1]);
}
void f4()
{
        printf("goto 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,i_);
        printf("%s = %s * 4\n",temp,st1[top]);
        strcpy(st1[top],temp);
        i_[0]++;
        strcpy(temp,"t");
        strcat(temp,i_);
        printf("%s = %s [ %s ] \n",temp,st1[top-1],st1[top]);
        top--;
        strcpy(st1[top],temp);
        i_[0]++;
}
void codegen()
{
        strcpy(temp,"t");
        strcat(temp,i_);
        printf("%s = %s %s %s\n",temp,st1[top-2],st1[top-1],st1[top]); top-=2;
        strcpy(st1[top],temp);
        i_[0]++;
}
void codegen_umin()
{
        strcpy(temp,"t");
        strcat(temp,i_);
        printf("%s = -%s\n",temp,st1[top]);
        top--;
        strcpy(st1[top],temp);
        i_[0]++;
}
void codegen_assign()
{
        printf("%s = %s\n",st1[top-2],st1[top]);
        top-=2;
}


%}
%token<ival> INT FLOAT VOID %token<str> ID NUM REAL LE GE EQ NEQ AND
OR %token WHILE IF RETURN PREPROC STRING PRINT FUNCTIONDOARRAY
ELSE STRUCT STRUCT_VAR FOR
%left LE GE EQ NEQ AND OR '<' '>' %right '='
```

```
%right UMINUS
%left '+' '-'
%left '*' '/'
%type<str> assignment assignment1 consttype '=' '+' '-' '*' '/'
ETF%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);
      }
      }
      ;

Type : INT
      | FLOAT
      | VOID
      ;

CompoundStmt : '{' StmtList '}'
      ;

StmtList : StmtList stmt
      |
      ;

stmt : Declaration
      | if
      | ID '(' ')' ';'
```

```
        | while
        | dowhile
        | for
        | RETURN consttype ';'
        | RETURN ';'
        | ';'
        | PRINT '(' STRING ')' ';'
        | CompoundStmt
        ;

dowhile : DO {dw1();} CompoundStmt WHILE '(' E ')' {dw2();} ';' ;

for : FOR '(' E {f1();} ';' E {f2();}';' E {f3();} ')' CompoundStmt {f4();};

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

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

while : WHILE {w1();}'(' E ')' {w2();} CompoundStmt {w3();} ;

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

assignment1 : ID {push($1);} '=' {strcpy(st1[++top],"=");} E{codegen_assign();}

        | ID ',' assignment1
        | consttype ',' assignment1
        | ID
        | consttype
        ;
consttype : NUM
        | REAL
        ;
```

```
Declaration : Type ID {push($2);} '=' {strcpy(st1[++top],"=");} E{codegen_assign();}
';'
        | assignment1 ';'
        | Type ID '[' assignment ']' ';'
        | ID '[' assignment1 ']' ';'
        | STRUCT ID '{' Declaration '}' ';'
        | STRUCT ID ID ';'
        | 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[])
{
        yyin =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;
}
```

2. Switch

Lex:

```
%{
#include<stdio.h>
#include "y.tab.h"
extern YYSTYPE yylval;
int flag=0;
int count = 0;
%}
%option yylineno
%%
"//"[' 'a-zA-Z0-9.]* ;
\/\*(.*\n)*.*\*\/ ;
int {ECHO; yylval.var_type=strdup(yytext);return INT;} float {ECHO;
yylval.var_type=strdup(yytext);return FLOAT;} char {ECHO;
yylval.var_type=strdup(yytext);return CHAR;} "case" { ECHO; return CASE;}
switch {ECHO;return SWITCH;}
default {ECHO;return DEFAULT;}
"'" {ECHO;return SINGLE;}
"," {ECHO;return COMMA;}
":" {ECHO;return COLON;}
";" {ECHO; return SC;}
[0-9]+((\.[0-9]+)?) {ECHO;yylval.text=strdup(yytext);return NUM;}"." {ECHO;
return DOT;}


"+=" {ECHO; yylval.text=strdup(yytext); return SPLUS;} "-=" {ECHO;
yylval.text=strdup(yytext); return SMINUS;} "*=" {ECHO;
yylval.text=strdup(yytext); return SMULT;} "/=" {ECHO;
yylval.text=strdup(yytext); return SDIV;} "++" {ECHO;
yylval.text=strdup(yytext); return INC;} "--" {ECHO; yylval.text=strdup(yytext);
return DEC;} "(" {ECHO;return OPEN;}
")" {ECHO;return CLOSE;}
"<=" {ECHO;yylval.text=strdup(yytext); return LESEQ; } ">="
{ECHO;yylval.text=strdup(yytext); return GRTEQ;} "!=" {ECHO;
yylval.text=strdup(yytext); return NOTEQ;} "==" {ECHO;
yylval.text=strdup(yytext); return EQEQ; } "<" {ECHO;
yylval.text=strdup(yytext); return LESS;}
">" {ECHO; yylval.text=strdup(yytext); return GREAT;} "{" {ECHO; return
OPBRACE;}
"}" {ECHO;return CLBRACE;}
"+" {ECHO;yylval.text=strdup(yytext); return PLUS;} "-"
{ECHO;yylval.text=strdup(yytext); return MINUS;}
"=" {ECHO;yylval.text=strdup(yytext); return ASSIGN; } "*"
{ECHO;yylval.text=strdup(yytext); return MULT;} "/"
{ECHO;yylval.text=strdup(yytext); return DIV;} "^"
```

```
{ECHO;yylval.text=strdup(yytext); return POW;} "||"
{ECHO;yylval.text=strdup(yytext); return OR;} "&&"
{ECHO;yylval.text=strdup(yytext); return AND;} [' '\n\t\s] {ECHO;}
continue {ECHO;return CONTINUE;} break {ECHO;return
BREAK;}
return {ECHO;return RETURN;}
if {ECHO;return IF;}
while {ECHO;return WHILE;}
for {ECHO;return FOR;}
else {ECHO;return ELSE;}
main {ECHO; return MAIN;}

[a-zA-Z]+[a-zA-Z0-9]* {ECHO;yylval.text=strdup(yytext);
return(ID);}# return 0;
%%

Yacc:
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <stdarg.h>
#include <ctype.h>
void yyerror(const char*);
int yylex();
int scope[100];
int scope_ctr;
int scope_ind;
int flag1 =1;

extern int yylineno;

int t=0;
int lab=0;
char stack[100][100];
int top=0;
int num_iter[10];
int iter_init[10];
int iter_top=0;
int number_of_iterations=0;
int return_flag=0;
typedef struct AST{
        char lexeme[100];
        int NumChild;
        struct AST **child;
}AST_node;
char typ[10]="nothing";
extern FILE *yyin;

/*
struct attributes{
        char* code;
        struct attributes* true;
        struct attributes* false;
        struct attributes* next;
        char addr[20];
}; */

char* code_gen(int arg_count,...);
char* gen_addr(char* string);
char* new_temp();
char* code_concatenate(int arg_count,...);
char* new_label();
int is_int(char *,int);
void remove_rest(char *, char *);
char break_lab[20];
char switch_id[10];
%}




%token DOT SINGLE SC COMMA LETTER OPBRACE CLBRACECONTINUE
BREAK IF ELSE FOR WHILE POWOPENCLOSECOMMENT SQ_OPEN
SQ_CLOSE

%union {char* var_type; char* text; struct AST *node; struct
        attributes{char* code;
        char* optimized_code;
        char* true;
        char* false;
        char* next_lab;
        char* next;
        char* addr;
        float val;
        int is_dig;
}A;}

%token <var_type> INT FLOAT CHAR %token <text> ID NUM PLUS MINUS
MULT DIV ANDORLESSGREATLESEQ GRTEQ NOTEQ EQEQ ASSIGN  SPLUS
SMINUS SMULTSDIVINC DEC SWITCH
%token <A> MAIN RETURN DEFAULT CASE COLON%type <var_type>
Type
%type <text> relOp logOp s_op
%type <A> F T E assign_expr1 assign_expr relexp logexp cond decl s_operation
unary_expr iter_stat stat comp_stat start jump_stat select_stat STCB D Varlist
%%



start:INT MAIN OPEN CLOSE comp_stat { //printf("Here\n"); $$=$5;
                                                                                                    if(return_flag)
                                                                                                { $$.code=
code_concatenate(2,$$.code,"end: ");

        $$.optimized_code = code_concatenate(2,$$.optimized_code,"end: "); }
                                                                                                    char* code1=
(char*)malloc(strlen($$.code));
                                                                                                    char* code2=
(char*)malloc(strlen($$.optimized_code));

                                                                                                        int a =
remove_blank($$.code,code1);
                                                                                                        //a =
remove_blank($$.optimized_code,code2);
                                                                                                  //printf("\nIC:
\n%s",code1);
                                                                                                  //printf("\nOC:
\n%s",code2);
```

```
                                                                                  char* code3=
(char*)malloc(strlen(code1));
                                                                  //char*code4=
(char*)malloc(strlen(code2));

      remove_rest(code1,code3);

      //remove_rest(code2,code4);
                                                                               //printf("\nIC:
\n%s",$$.code);
                                                                               //printf("\nIC:
\n%s",code1);
                                                                               //printf("\nOC:
\n%s",code2);

      //printf("\n\nOC: %s\n\n",$$.optimized_code);
                                                                printf("\n\nIC\n\n");
                                                                               print_IC(code3);
                                                                               YYACCEPT;
                                                                                         }
   ;

comp_stat: OPBRACE {scope_ctr++;scope[scope_ind++]=scope_ctr;}stat CLBRACE
{$$=$3; scope[scope_ind++]=0; $$.optimized_code =$3.optimized_code;}
              ;

stat:E SC stat {$$.code = code_concatenate(2,$1.code,$3.code); $$.optimized_code =
code_concatenate(2,$1.optimized_code, $3.optimized_code);}
   | assign_expr stat {$$.code = code_concatenate(2,$1.code,$2.code); //printf("$2
                                             optimized code: %s\n\n",
$2.optimized_code);
                                                                          $$.optimized_code =
code_concatenate(2,$1.optimized_code, $2.optimized_code);
                                             $$.is_dig=$1.is_dig;
                                                           //printf("ABC\n");
                                        }
   |comp_stat stat {$$.code = code_concatenate(2,$1.code,$2.code);
$$.optimized_code = code_concatenate(2,$1.optimized_code, $2.optimized_code);}
   |{char * lab = new_label(); push(lab);} select_stat {pop();} stat {$$.code=
code_concatenate(2,$2.code,$4.code);
```

```
                                                            $$.optimized_code =
code_concatenate(2,$2.optimized_code, $4.optimized_code);

                                                         }
   | {char * lab = new_label(); push(lab);} iter_stat {pop();} stat {$$.code=
code_concatenate(2,$2.code,$4.code);

                                                            $$.optimized_code =
code_concatenate(2,$2.optimized_code, $4.optimized_code);

                                                         }
   | jump_stat stat {$$.code = code_concatenate(2,$1.code,$2.code);
$$.optimized_code = code_concatenate(2,$1.optimized_code, $2.optimized_code);}
   |decl stat {$$.code = code_concatenate(2,$1.code,$2.code);
$$.optimized_code = code_concatenate(2,$1.optimized_code,
$2.optimized_code);}
   | {$$.code = " "; $$.optimized_code = $$.code;} ;



                                             ST : SWITCH OPEN ID CLOSE OPBRACE
                              {scope_ctr++;scope[scope_ind++]=scope_ctr;} B CLBRACE{

        scope[scope_ind++]=0;

        if(!look_up_sym_tab($3)){printf("Undeclared variable %s\n", $3);
YYERROR;}
                                                                   $$.code
= code_concatenate(2, $7.code, code_gen(2,stack[top],": "));

        $$.optimized_code = code_concatenate(2, $7.optimized_code,
code_gen(2,stack[top],": "));
                                                              }
   ;


B : C {$$=$1;}
   | C D {$$.code = code_concatenate(2,$1.code, $2.code);
$$.optimized_code = code_concatenate(2,$1.optimized_code,
$2.optimized_code);}
   | C B {$$.code = code_concatenate(2,$1.code, $2.code);
$$.optimized_code = code_concatenate(2,$1.optimized_code,
$2.optimized_code);}
   ;

C : CASE NUM COLON stat {char* lab1 = new_label(); //printf("in switch\n");
                                                          char* lab2 = new_label();
                                                         //printf("in switch\n");
                                                                     $$.code =
  code_concatenate(6,code_gen(4,"if ",$<text>-2," == ",$2," goto ", lab1), "goto",lab2,
         code_gen(2,lab1,": "),$4.code, code_gen(2,lab2,": ")); //printf("in switch\n");
                                                                $$.optimized_code
=code_concatenate(6,code_gen(6,"if ",$<text>-2," == ",$2," goto ", lab1), "goto ",lab2,
code_gen(2, lab1,": "),$4.optimized_code, code_gen(2,lab2,": "));

     //printf("%s\n",$4.optimized_code);
                                                             }
   ;


D : DEFAULT COLON stat {char* lab = new_label(); $$.code
=code_concatenate(1,$3.code); $$.optimized_code = $3.optimized_code;};




select_stat: ST {$$=$1;}
             ;

iter_stat:FOR OPEN decl cond SC E CLOSE comp_stat
{char*begin=new_label();
            $3.next = begin;

            $4.true = new_label();



            $8.next = stack[top];
```

```
        //push($8.next);

        $4.false = $8.next;

        //printf("here %s",break_lab);

        strcpy(break_lab,$8.next);

        //printf("here2 %s",break_lab);

        //$8.next = new_label();

        $6.next = begin;


        $$.code = code_concatenate(8,$3.code, code_gen(2,begin,": 
"),code_gen(4,"if ", $4.code, "goto ", $4.true),

        code_gen(2,"goto ", $4.false), code_gen(2,$4.true,": "),$8.code, 
code_gen(2,"goto ",begin),code_gen(2,$8.next,": \n"));


        if(num_iter)

        {

            //printf("HERE\n");

            $$.optimized_code = " ";

            $$.optimized_code = 
code_concatenate(2,$$.optimized_code,$3.optimized_code);
            //printf("number of iterations %d\n",number_of_iterations);

            int i;

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

            {
```

```
                                                   $$.optimized_code =
        code_concatenate(2,$$.optimized_code,$8.optimized_code); }

                                   //printf("FOR OC%s\n",$$.optimized_code);

             }

             else

             {

                   $$.optimized_code = $$.code;

             }

             }
             ;

jump_stat: BREAK SC {$$.code = code_gen(2,"goto ",stack[top]);
 $$.optimized_code = $$.code;}
             |RETURN E SC {$$.code = code_gen(3,"return
",$2.addr,"\ngotoend\n"); $$.optimized_code = $$.code; return_flag=1;} ;

cond:relexp {$$=$1;}
        |logexp {$$=$1;}
        |E {$$=$1;}
        ;


relexp:E relOp E {$$.code =
code_gen(3,$1.code,$3.code,code_gen(3,$1.addr,$2,$3.addr));
                                  if(strcmp($2,"<")==0)
                                  {
                                                   num_iter[iter_top++]=atoi($3.addr);
                                                   //printf("TOP: %d\n",iter_top);
                                                      number_of_iterations = num_iter[--
iter_top]-iter_init[iter_top];
                                                   //num_iter = atoi($3.addr);
                                                      //printf("Number of
iterations: %d\n",num_iter);
                                        }

                                                           if(strcmp($2,"<=")==0)
                                        {

                                                               num_iter[iter_top++]=atoi($3.addr);
                                                               printf("TOP: %d\n",iter_top);
                                                                   number_of_iterations = num_iter[--
iter_top]-iter_init[iter_top]+1;
                                        }
                                                           if(strcmp($2,">")==0)
                                        {

                                                               num_iter[iter_top++]=atoi($3.addr);
                                                               printf("TOP: %d\n",iter_top);
                                                       number_of_iterations = iter_init[--iter_top]-
num_iter[iter_top];
                                        }
                                                           if(strcmp($2,">=")==0)
                                        {

                                                               num_iter[iter_top++]=atoi($3.addr);
                                                               printf("TOP: %d\n",iter_top);
                                                       number_of_iterations = iter_init[--iter_top]-
num_iter[iter_top]+1;
                                        }

                                        }
          ;
logexp:E logOp E {$$.code =
code_gen(3,$1.code,$3.code,code_gen(3,$1.addr,$2,$3.addr));
$$.optimized_code = $$.code;}
          ;

logOp:AND {$$=$1;}
         |OR {$$=$1;}
         ;

relOp:LESEQ {$$=$1;}
    |GRTEQ {$$=$1;}
    |NOTEQ {$$=$1;}
    |EQEQ {$$=$1;}
```

```
        |LESS {$$=$1;}
        |GREAT {$$=$1;}
        ;

decl:Type Varlist SC {$$.code = code_gen(3,$1," ",$2.code);
$$.optimized_code = $$.code;}
   |Type assign_expr1 {$$.code = code_gen(1,$2.code);
$$.optimized_code=$2.optimized_code; }

   ;


Type:INT { $$=$1; strcpy(typ,$1);} |FLOAT
        {$$=$1;strcpy(typ,$1);}
        ;

   Varlist:Varlist COMMA ID {if(look_up_sym_tab_dec($3,scope[scope_ind-1])){
                        yyerror("Redeclaration\n"); YYERROR; } if(scope[scope_ind-
1]>0){update_sym_tab($<var_type>0,$3,yylineno,scope[scope_ind- 1]);}else{int
scop=get_scope();update_sym_tab($<var_type>0,$3,yylineno,scop);}

                                                $$.code = code_gen(3,$1.code,",
",$3); $$.optimized_code = code_gen(3,$1.optimized_code,", ",$3); }
          |ID {$$.addr=gen_addr($1);
        if(look_up_sym_tab_dec($1,scope[scope_ind-
1])){ yyerror("Redeclaration\n"); YYERROR; } if(scope[scope_ind-
1]>0){update_sym_tab($<var_type>0,$1,yylineno,scope[scope_ind- 1]);}else{int
scop=get_scope();update_sym_tab($<var_type>0,$1,yylineno,scop);}$$.code = $1;
                                    $$.optimized_code=$1;
                                               //printf("%s ID\n",$1);
                                                       }

          ;

assign_expr:ID ASSIGN E COMMA assign_expr {$$.addr =gen_addr($1);$$.code =
code_concatenate(3,$3.code,code_gen(3,$$.addr," = ",$3.addr),$5.code); char
                                                       buf[10];
                                                       int scop=get_scope();
                                                                  int
ret=is_int($1,scop);
```

```
                                                                                    if(ret)
                                                                                           {

      if(is_digit($3.addr))
                                                                                                 {
                                                                                             float
val=atof($3.addr);
                                                                                             int
val1=(int)val;
                                                                                char
buf1[10];

      gcvt(val1, 6, buf1);
                                                                               $$.code
= code_concatenate(3,$3.code,code_gen(3,$$.addr," = ",buf1),$5.code); }
                                                                         else
                                                                               $$.code
= code_concatenate(3,$3.code,code_gen(3,$$.addr," = ",$3.addr),$5.code); }
                                                                   else
                                                                          $$.code=
code_concatenate(3,$3.code,code_gen(3,$$.addr," = ",$3.addr),$5.code);
                                                                   //printf("%s\n",$1);

                                                                                    if(ret)
                                                                                           {
                                                                                       int
val=(int)$3.val;
                                                                                    float f =val;
                                                                                      gcvt(val, 6,
buf);
                                                                                           }
                                                                                         else
                                                                                   gcvt($3.val, 6,
buf);
                                                                                    if($3.is_dig)
                                                                    $$.optimized_code=
code_concatenate(2,code_gen(3,$$.addr," = ",buf),$5.optimized_code); else
                                                                    $$.optimized_code=
code_concatenate(3,$3.optimized_code,code_gen(3,$$.addr,"
=",$3.addr),$5.optimized_code);
```

```
        if(!look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}flag1=1;
                                                                                                                        }
                  |ID ASSIGN E SC {
                                                                                                            $$.addr =
gen_addr($1);
                                                                                         int scop=get_scope();
                                                                                                          int
ret=is_int($1,scop);
                                                                                                             if(ret)
                                                                                                                        {

        if(is_digit($3.addr))
                                                                                                                                {
                                                                                                                            float
val=atof($3.addr);
                                                                                                                            int
val1=(int)val;
                                                                                                         char
buf1[10];

        gcvt(val1, 6, buf1);
                                                                                                         $$.code
= code_concatenate(2,$3.code,code_gen(3,$$.addr," = ",buf1)); }
                                                                                                 else
                                                                                                         $$.code
= code_concatenate(2,$3.code,code_gen(3,$$.addr," = ",$3.addr)); }
                                                                                                                     else
                                                                                                  $$.code=
code_concatenate(2,$3.code,code_gen(3,$$.addr," = ",$3.addr)); char buf[10];


                                                                                                             if(ret)
                                                                                                                        {
                                                                                                                 int
val=(int)$3.val;
                                                                                                             float f =val;
                                                                                                               gcvt(val, 6,
buf);
```

```
                                                                                    }
                                                                                  else
                                                                            gcvt($3.val, 6,
buf);

        //printf("flag %d",flag1);
                                                                            if($3.is_dig)
                                                                                    {

        //printf("Assign Here\n");

                     $$.optimized_code = code_concatenate(1,code_gen(3,$$.addr," =",buf));

        //printf("%s\n",$$.optimized_code);
                                                                                    }
                                                                                  else
                                                              {
        $$.optimized_code =
code_concatenate(2,$3.optimized_code,code_gen(3,$$.addr," =",$3.addr)); //flag1=1;
                                                                                    }

        //printf("OC: %s\n",$$.optimized_code);

        if(!look_up_sym_tab($1)){printf("Undeclared variable %s\n", $1);
YYERROR;}
                                                                          flag1=1;

                                                                                    }
                    ;

assign_expr1:ID ASSIGN E COMMA assign_expr1 {$$.addr =gen_addr($1);


        if(strcmp(typ,"int")==0)
                                                                                    {

        if(is_digit($3.addr))
                                                                                         {
                                                                                     float
val=atof($3.addr);
```

```
                                                                                      int
val1=(int)val;
                                                                        char
buf1[10];

      gcvt(val1, 6, buf1);
                                                                        $$.code
= code_concatenate(3,$3.code,code_gen(5,typ," ",$$.addr," = ",buf1),$5.code);}
                                                                  else
                                                                        $$.code
= code_concatenate(3,$3.code,code_gen(5,typ," ",$$.addr," = ",$3.addr),$5.code);

                                                                                    }
                                                            else
```