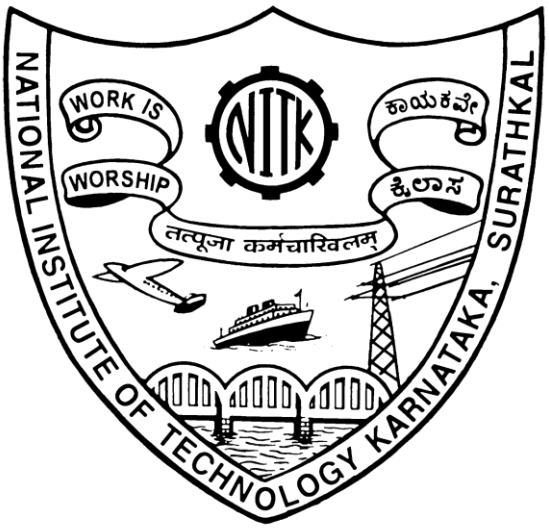
Intermediate code generation for the C Language



National Institute of Technology Karnataka Surathkal

Date: 29/03/2017

Submitted To: Uma Priya

Group Members:

Sheetal Shalini (14CO142)

Navya R S (14CO126)

Abstract:

A **compiler** is a computer program that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or passes, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Lexical analysis, Syntax analysis, Semantic analysis and Intermediate Code generation are the four phases of the frontend of the compiler. Intermediate code generation phase is the final phase of compiler front-end. Its main aim is to convert the program into a format expected by the compiler back-end. Output of the semantic phase is the input to this phase. Generally this phase is followed by Intermediate code optimisization and then machine code generation. Intermediate codes are machine independent codes, but they are close to machine instructions. The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator. Intermediate code generator eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers. If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. In this project, for assignments and expressions, every variable declared and result of every expression is stored in a temporary variable generated. Conditional and iterative statements are handled by goto statements and labels. Precedence to the operators are given so that the computation is done accordingly. Three address code is produced even for arrays.

Contents:

Introduction

Intermediate code generation

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

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

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

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

Flex Script

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

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

Definition section

%%

Rules section

%%

C code section

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

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

Yacc Script

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

declarations

%%

rules

%%

routines

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

THE DECLARATIONS SECTION may contain the following items.

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token.
- Declaration of the start symbol using the keyword %start
- C declarations: included files, global variables, types.
- C code between %{ and %}.

RULES SECTION. A rule has the form:

nonterminal : sentential form

| sentential form

.....

| sentential form

;

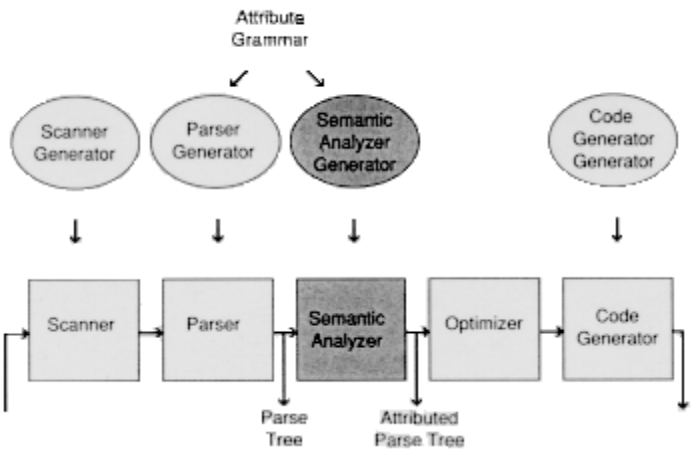
Actions may be associated with rules and are executed when the associated sentential form is matched. Each sentential form has an action associated with it which describes the semantic rules for type mismatch, handling undeclared variables, redeclaration, scope of variables. Also each production has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

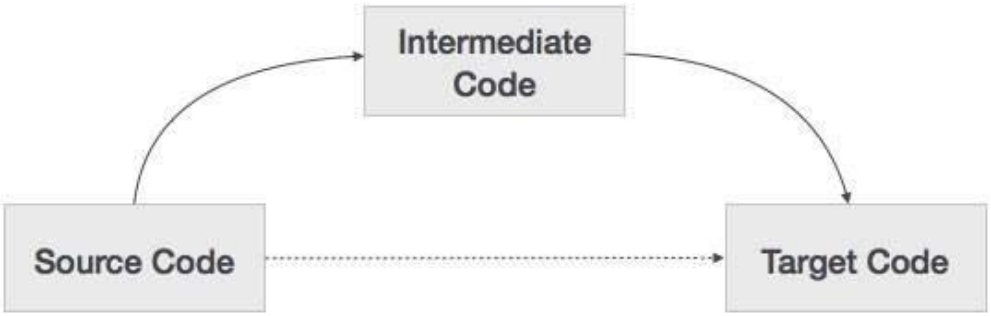
C Program

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

Design of Programs

Flow :





Compiler built checks for syntax and semantic errors and displays them if exists and then generates three address codes for error free programs. Three-address codes are generated and the temparory variables produced are displayed.

Codes:

1. Lex file (parser.l) :

```
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;
printf  return PRINT;
{alpha}({alpha}|{digit})* {yylval.str=strdup(yytext); return ID;}
{digit}+    {yylval.str=strdup(yytext);return NUM;}
{digit}+\.{digit}+  {yylval.str=strdup(yytext); return REAL;}
"<="        return LE;
">="        return GE;
"=="        return EQ;
"!="        return NEQ;
"&&"        return AND;
"||"        return OR;
\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
\".*\"  return STRING;
.   return yytext[0];
%%
```

2. Yacc file (parser.y) :

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include "s_symbol.c"
    int g_addr = 100;
int i=1;
int
stack[100],index1=0,end[100],arr[10],gl1,gl2,ct,c,b,fl,top=0,label[20],lnum
=0,ltop=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 FUNCTION DO ARRAY ELSE STRUCT
STRUCT_VAR FOR
%right '='
%left UMINUS
%type<str> assignment assignment1 consttype '=' '+' '-' '*' '/' E T F
%type<ival> Type
%union {
        int ival;
        char *str;
    }
%%

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

Function : Type ID '('')' CompoundStmt {
    if ($1!=returntype_func(ct))
    {
        printf("\nError : Type mismatch : Line %d\n",printline());
    }

    if (!(strcmp($2,"printf") && strcmp($2,"scanf") && strcmp($2,"getc") &&
strcmp($2,"gets") && strcmp($2,"getchar") && strcmp ($2,"puts") &&
strcmp($2,"putchar") && strcmp($2,"clearerr") && strcmp($2,"getw") &&
```

```
strcmp($2,"putw") && strcmp($2,"putc") && strcmp($2,"rewind") &&
strcmp($2,"sprint") && strcmp($2,"sscanf") && strcmp($2,"remove") &&
strcmp($2,"fflush")))
        printf("Error : Type mismatch in redeclaration of %s : Line
%d\n",$2,printline());
    else
    {
        insert($2,FUNCTION,g_addr);
        insert($2,$1,g_addr);
        g_addr+=4;
    }
    }
    ;

Type : INT
    | FLOAT
    | VOID
    ;

CompoundStmt : '{' StmtList '}'
    ;

StmtList : StmtList stmt
    |
    ;

stmt : Declaration
    | if
    | while
    | dowhile
    | for
    | RETURN consttype ';' {
                    if(!(strspn($2,"0123456789")==strlen($2)))
                        storereturn(ct,FLOAT);
                    else
                        storereturn(ct,INT); ct++;
                    }
    | RETURN ';' {storereturn(ct,VOID); ct++;}
    | RETURN E ';'
    | ';'
    | 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();}
    {
        int sct=returnscope($1,stack[index1-1]);
        int type=returntype($1,sct);
        if((!(strspn($5,"0123456789")==strlen($5))) && type==258 && fl==0)
            printf("\nError : Type Mismatch : Line %d\n",printline());
        if(!lookup($1))
        {
            int currscope=stack[index1-1];
            int scope=returnscope($1,currscope);
            if((scope<=currscope && end[scope]==0) && !(scope==0))
            {
                check_scope_update($1,$5,currscope);
            }
        }
        }

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

consttype : NUM
    | REAL
    ;

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

```
            else
            {
                insert_dup($2,$1,g_addr,currscope);
                check_scope_update($2,$6,stack[index1-1]);
                int sg=returnscope($2,stack[index1-1]);
                g_addr+=4;
            }
        }
        else
        {
            int scope=stack[index1-1];
            insert($2,$1,g_addr);
            insertscope($2,scope);
            check_scope_update($2,$6,stack[index1-1]);
            g_addr+=4;
        }
    }

    | assignment1 ';'  {
                if(!lookup($1))
                {
                    int currscope=stack[index1-1];
                    int scope=returnscope($1,currscope);
                    if(!(scope<=currscope && end[scope]==0) || scope==0)
                        printf("\nError : Variable %s out of scope : Line
%d\n",$1,printline());
                }
                else
                    printf("\nError : Undeclared Variable %s : Line
%d\n",$1,printline());
                }

    | Type ID '[' assignment ']' ';' {
                        insert($2,ARRAY,g_addr);
                        insert($2,$1,g_addr);
                        g_addr+=4;
                    }
    | ID '[' assignment1 ']' ';'
    | STRUCT ID '{' Declaration '}' ';' {
                        insert($2,STRUCT,g_addr);
                        g_addr+=4;
                        }
    | STRUCT ID ID ';' {
                insert($3,STRUCT_VAR,g_addr);
                g_addr+=4;
                }
    | error
    ;

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

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

```
   | ID {push($1);} AND {strcpy(st1[++top],"&&");} E {codegen();}
   | ID {push($1);} OR {strcpy(st1[++top],"||");} E {codegen();}
   | ID {push($1);} '<' {strcpy(st1[++top],"<");} E {codegen();}
   | ID {push($1);} '>' {strcpy(st1[++top],">");} E {codegen();}
   | ID {push($1);} '=' {strcpy(st1[++top],"||");} E {codegen_assign();}
   | array {array1();}
   ;
T : T '*'{strcpy(st1[++top],"*");} F{codegen();}
   | T '/'{strcpy(st1[++top],"/");} F{codegen();}
   | F
   ;
F : '(' E ')' {$$=$2;}
   | '-'{strcpy(st1[++top],"-");} F{codegen umin();} %prec UMINUS
   | ID {push($1);fl=1;}
   | NUM {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;
}
```

Explanation:

Files :

1. **Parser.l** : Lex file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.
2. **Parser.y** : Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned. This fils also contains the semantic rules defined

against every production necessary. Rules for producing three address code is also present against the productions and in functions.

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

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

Yacc file has productions to check the following functionalities:

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

Test Cases:

1. Code (arithmetic expressions) :

```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    int c=7;
    int d=a+b-c*a/d;
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test1.c
a = 5
b = 6
c = 7
t0 = a + b
t1 = c * a
t2 = t1 / d
t3 = t0 - t2
d = t3
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     6       1       INT
3       c       108     7       1       INT
4       d       112     0       1       INT
5       main    116     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

2. Code (if-else statement) :

```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    if(a<=7)
    {
        b=b-4;
    }
    else
    {
        b=2;
    }
    return 0;
}
```

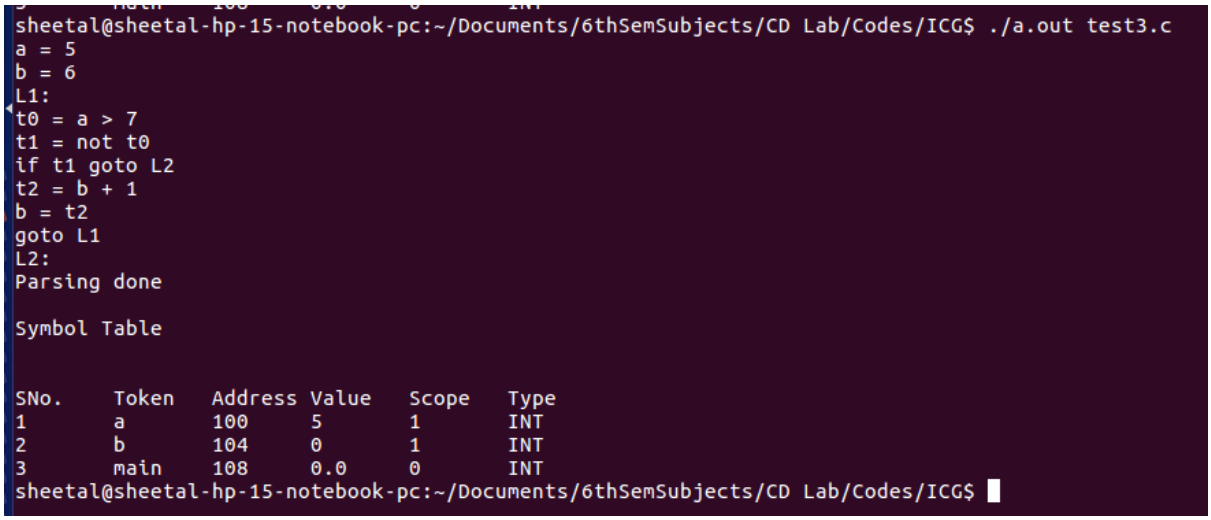
Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test2.c
a = 5
b = 6
t0 = a <= 7
t1 = not t0
if t1 goto L1
t2 = b - 4
b = t2
goto L2
L1:
b = 2
L2:
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     2       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

3. Code (while loop) :

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

Output :



4. Code (for loop) :

```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    for(a=9;a!=6;a=a-1)
    {
        b=b+4;
        b=2;
    }
    b=b/9;
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test4.c
a = 5
b = 6
a = 9
L1:
t0 = a != 6
t1 = not t0
if t1 goto L2
goto L3
L4:
t2 = a - 1
a = t2
goto L1
L3:
t3 = b + 4
b = t3
b = 2
goto L4
L2:
t4 = b / 9
b = t4
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     0       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

5. Code (do-while loop) :

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

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test5.c
a = 5
b = 6
L1:
t0 = b + 1
b = t0
t1 = a > 7
if t1 goto L1
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     0       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

6. Code (nested if-else statement) :

```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    if(a<=7)
    {
        if(a==9)
        {
            b=b*8;
            b=9;
        }
        else
        {
            a=10;
        }
    }
    else
    {
        b=2;
    }
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test6.c
a = 5
b = 6
t0 = a <= 7
t1 = not t0
if t1 goto L1
t2 = a == 9
t3 = not t2
if t3 goto L2
t4 = b * 8
b = t4
b = 9
goto L3
L2:
a = 10
L3:
goto L4
L1:
b = 2
L4:
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     10      1       INT
2       b       104     2       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

7. Code (nested while loop) :

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

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test7.c
a = 5
b = 6
L1:
t0 = a > 7
t1 = not t0
if t1 goto L2
b = 6
L3:
t2 = b >= 5
t3 = not t2
if t3 goto L4
a = 9
goto L3
L4:
goto L1
L2:
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     9       1       INT
2       b       104     6       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

8. Code (nested for loop) :

```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    for(a=9;a!=6;a=a-1)
    {
        b=2;
        for(b=0;b<=9;b=b*2)
        {
            b=7;
        }
    }
    b=b/9;
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test8.c
a = 5
b = 6
a = 9
L1:
t0 = a != 6
t1 = not t0
if t1 goto L2
goto L3
L4:
t2 = a - 1
a = t2
goto L1
L3:
b = 2
b = 0
L5:
t3 = b <= 9
t4 = not t3
if t4 goto L6
goto L7
L8:
t5 = b * 2
b = t5
goto L5
L7:
b = 7
goto L8
L6:
goto L4
L2:
t6 = b / 9
b = t6
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     0       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

9. Code (nested do-while loop) :

```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    do
    {
        b=b+1;
        do
        {
            b=9;
        }while(a==7);
    }while(a>7);
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test9.c
a = 5
b = 6
L1:
t0 = b + 1
b = t0
L2:
b = 9
t1 = a == 7
if t1 goto L2
t2 = a > 7
if t2 goto L1
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       a       100     5       1       INT
2       b       104     9       1       INT
3       main    108     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

10. Code (Arrays) :

```
#include <stdio.h>
int main()
{
    int ab=3;
    int a[10];
    int x=a[ab-2];
    return 0;
}
```

Output :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$ ./a.out test10.c
ab = 3
t0 = ab - 2
t1 = t0 * 4
t2 = a [ t1 ]
x = t2
Parsing done

Symbol Table


SNo.    Token   Address Value   Scope   Type
1       ab      100     3       1       INT
2       a       104     0.0     0       INT
3       x       108     0       1       INT
4       main    112     0.0     0       INT
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/CD Lab/Codes/ICG$
```

Implementation

1. The functions for while loop ICG :

```
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);
}
```

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

2. Functions for if-else condition statement :

```
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--]);
}
```

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

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

```
    ;
```

3. Function for array ICG :

```
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]++;
}
```

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

These functions have been implemented in parser.y file.

Results :

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

lex parser.l

yacc parser.y

gcc y.tab.c -ll -ly

./a.out test.c

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

Future work :

Intermediate code generator generates three address codes for:

1. assignments

2. expressions

3. arrays

4. conditional statements

5. iterative statements

We will try to extend it to pointers and structures.

References :

1. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

2. https://web.cs.wpi.edu/~kal/

3. https://www.tutorialspoint.

4. http://www.iith.ac.in/~ramakrishna/Compilers-Aug14/slides/17-intermediate-CG-1.pdf

5.https://gradeup.co/intermediate-code-generation-i-b54685df-c4e9-11e5-b8bc-a664533bc498