Name: Gaurav Thakkar
Class: CSE 1
Batch: A
Roll No: 2183505

Assignment 6

TITLE: Implement Intermediate Code generation for sample language using LEX and YACC.

PROBLEM STATEMENT:

Write a Program for Intermediate Code Generation for subset of C (If loop) using LEX & YACC

OBJECTIVES:

1) To understand the different forms of ICG
2) To understand how a compiler ICG phase works for subset of C.

THEORY:

Syntax trees and postfix notation are two kinds of intermediate representations.
A third called three address code will also be used.
The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax
trees for generating postfix notation.

6.2.1 Graphical Representations :

A syntax tree depicts the natural hierarchical structure of a source program.
A dag gives the same information but in compact way because common subexpressions are identified.
A syntax tree and dag for the assignment statement
a : = b * − c + b * − c appear in
Fig. 6.2.1.

(a) Syntax tree

(b) DAG

Fig. 6.2.1 : Graphical representation at a : = b * − c + b * − c

Postfix notation is a linearized representation of a syntax tree, it is a list of the
nodes of a tree in which a node appears immediately after its children.
The postfix notation for the syntax tree in Fig. 6.2.1(a) is
a bc uminus * bc uminus the *+ assign

Three · Address Code :

Three address code is a sequence of statement of the general form

x : y op z

where x, y and z are names constants or compiler generated temporaries.
op stands for any operator, such as fixed-or-floating point arithmetic operator
or logical operator on Boolean valued data.
No built up arithmetic expressions are permitted, as there is only one operator
on the right side of a statement.
A source language expression like x + y * z might be translated into a sequence. t1 : y * z

t2 : x + t1

where t1, t2 are compiler-generated temporary names.
Three –address code is a linearized representation of a syntax tree or a dag
in which explicit names correspond to the interior nodes of the graph.
The syntax tree and dag in Fig. 6.2.1 are represented by the three-address code
sequence in Fig. 6.2.4.Variables names can appear directly in three-addres
statements so Fig. 6.2.4(a) has no statements corresponding to the leaves in Fig.

Implementations of Three-Address Statements :

A three-address is an abstract form of intermediate form and these statements
can be implemented as records with fields for the operator and the operands.

There are three representations

a) Quadraple
b) Triples
c) Indirect triples

a) Quadraples :

A quadraple is a record structure with four fields, which we will call as op arg1, arg2 and result.

The op field contains an internal code for the operator.

The three-address statement x : = y op 2 is represented by placing y in arg1, z in arg2, and x in result.

Statements with unary operators like x : = − y or x : = y do not use arg 2.

Operator like param use neigher arg2 not result.

Conditional and unconditional jumps put the target label in result.

The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. So, temporary names must be entered into the symbol table as they are created.

The quadraples for the assignment a : b * − c + b * − c are shown in Fig..

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	B	t3	t4
(4)	+	t2	t4	t5
(5)	: =	t5		a

Fig.: Quadraples representation of three-address statement

b) Triples :

Triples is a record structure with three field arg1, arg2 and op.

The fields arg1 and arg2 for the arguments of op are either pointers to the symbol table or pointers into the triple structure.

Triples corresponds to the representation of a syntax tree or dag by an array of nodes.

Parenthesized numbers represent pointers into the triple structure. While symbol-table pointers are represented by the names themselves.

Fig. shows the triples for the assignment statement a : = b * − c + b * − c.

	op	Arg1	arg2

(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Fig.: Triple representation of three-address statement

Fig. shows more representations for the ternary operation like x [i] : = y, it requires two entries in the triple structure, while x : = y [i] is naturally represented as two operations.

	op	arg1	arg2			op	arg1	arg2
(0)	[ ] =	x	i		(0)	= [ ]	y	i
(1)	assign	(0)	y		(1)	assign	x	(0)

(a) x [i] : = y **(b) x : y [i]**

Fig. 6.2.9 : More triple representations

c) Indirect triples :

Indirect triple representation is the listing pointers to triples rather-than listing the triples themselves.

Let us use an array statement to list pointers to triples in the desired order.

Then the triples in Fig. 6.2.8 be represented as in Fig. 6.2.10.

	Statement			**op**	**arg1**	**arg2**
(0)	(14)		(14)	uminus	c	
(1)	(15)		(15)	*	b	(14)
(2)	(16)		(16)	uminus	c	
(3)	(17)		(17)	*	b	(16)
(4)	(18)		(18)	+	(15)	(17)
(5)	(19)		(19)	assign	a	(18)

Fig.: Indirect triples representation of three address statements

IMPLEMENTATION DETAILS / DESIGN LOGIC:

LEX

1. Declaration of header files specially y.tab.h which contains declaration for the tokens **RELOP ID IF BLST BLEND NUM A**
2. Define the Regular Expression for the tokens **RELOP ID IF BLST BLEND NUM A.**
3. End of declaration section with %%
4. Match regular expression.
5. If match found then store it in yylval
6. Return the token.

7. End rule-action section by %%
8. End

```
//******************lex specification*************//



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

%%
[/t]+ ;
if { strcpy(yylval.cval,yytext);return IF;}
[{] { strcpy(yylval.cval,yytext);return BLST;}
[}] { strcpy(yylval.cval,yytext);return BLEND;}
[0-9]+ {yylval.dval=atoi(yytext); return NUM;}
[a-z]+ {strcpy(yylval.cval,yytext);return ID;}
"+"|"-"|"="|"*"|"/" {return *yytext;}
"("|")" {return *yytext;}
"<"|">" {strcpy(yylval.cval,yytext);return RELOP;}
\n {return *yytext;}
%%

yywrap()
{
return 1;
}
```

YACC

1. Declaration of header files
2. Declare structure for three address code with fields for operator 1 , operator 2
   operand , temporary variable.
3. Declare tokens **RELOP ID IF BLST BLEND NUM A**
4. End of declaration section
5. Declaration of grammar
6. End the section by %%
7. Declare main() function
   a. Call yyparse() function .
8. End of the program.

```
//**********************yaccspecification***************//

%{
      #include<stdio.h>
      #include<math.h>
      #include<string.h>

struct quad
{

char op[10];
char arg1[10];
char arg2[10];
char res[10];
} quad[20];

int blk_cnt=1,cnt=1;
char arr[10],str[10],temp[10];
struct block
{
  int st,end,if_flag;
} blk[20];

%}

%union
{
    char cval[10];
    int dval;
}
%token <cval> RELOP ID IF BLST BLEND
%token <dval> NUM
%type <cval> A
%left '-' '+'
%left '*' '/'
%left RELOP


%%
S : B '(' E ')' S1 '\n'  {print();}
  ;
B : IF {blk[blk_cnt].if_flag=1;}
  ;
E : ID RELOP ID {strcpy(quad[cnt].op,$2);
                 strcpy(quad[cnt].arg1,$1);
                 strcpy(quad[cnt].arg2,$3);
                 arr[0]=arr[0]+1;
                 arr[1]='\0';
                 strcpy(str,"t");
                 strcpy(quad[cnt].res,str);
                 cnt++;
                }
   ;

S1 : BLST1 AS BLEND1
    ;
AS : AS1
    ;
AS1 : ID '=' A {  strcpy(quad[cnt].op,"=");
                 strcpy(quad[cnt].arg1,$3);
                 strcpy(quad[cnt].arg2,"-");
                 strcpy(quad[cnt].res,$1);
                 cnt++;
                }
     ;

A : ID {strcpy($$,$1);}
   | NUM {sprintf(temp,"%d",$1); strcpy($$,temp);}
   | A '+' A { strcpy(quad[cnt].op,"+");
                 strcpy(quad[cnt].arg1,$1);
                 strcpy(quad[cnt].arg2,$3);
                 arr[0]=arr[0]+1;
                 arr[1]='\0';
                 strcpy(str,"t");
                 strcat(str,arr);
                 strcpy(quad[cnt].res,str);
                 cnt++;
                 strcpy($$,str);
                }
   ;

BLST1 : BLST { if(blk[blk_cnt].if_flag==1)
                {
                     blk[blk_cnt].st=cnt;
                     strcpy(quad[cnt].op,"if");
                     strcpy(str,"t");
                 strcat(str,arr);
                 strcpy(quad[cnt].arg1,str);
                 temp[0]=cnt+'2';
                 temp[1]='\0';
                  strcpy(quad[cnt].res,temp);
                 cnt++;
                  strcpy(quad[cnt].op,"goto");
                 cnt++;
                 }
```

```
                  }
        ;
BLEND1 : BLEND { if(blk[blk_cnt].if_flag==1)
                   {
                      blk[blk_cnt].if_flag=0;
                      blk[blk_cnt].end=cnt;
                      temp[0]=cnt+'0';
                      temp[1]='\0';

strcpy(quad[blk[blk_cnt].st+1].res,temp);
                    }
                 blk_cnt++;
               }
          ;


%%
main()
{
yyparse();
}
yyerror()
{
 printf("ERROR");
 return 1;
}

print()
{
 int i;
 printf("\nS.No\tOpr\tArg1\tArg2\tResult\n");
 printf("==============================================");
 for(i=1;i<cnt;i++)
 {
   printf("\n%d\t%s\t%s\t%s\t%s",i,quad[i].op,quad[i].arg1,
quad[i].arg2,quad[i].res);
 }
 printf("\n %d",i);
}
```

TESTING

INPUT :

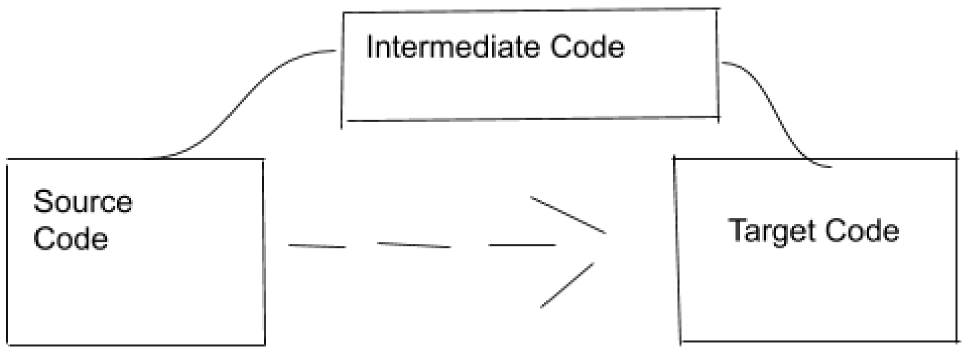
```
if(a<b) {a=b+1;};
```

OUTPUT

```
S.No    Opr     Arg1    Arg2    Result
=======================================================
1       <       a       b       t
2       if      ta      -       4
3       goto    -       -       6
4       +       b       1       tb
5       =       tb      -       a
6
```

FAQ's

1.) **What is the role/ significance of ICG in compilation?**



- If a compiler translates the source language to the target machine language without having the option to generate ICG then for each new machine, a full native code is required
- Intermediate Code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers
- It becomes easy to apply the source code modifications to

improve code performance by applying code optimisation techniques on the intermediate code

2.) **What are the different linear and graphical representations of Intermediate Code?**

Linear:
Pseudo Code for an abstract machine
Level of abstraction varies
Simple, compact Data Structures
Easier to rearrange
Ex 1. 3 address code
2. Stack Machine Code

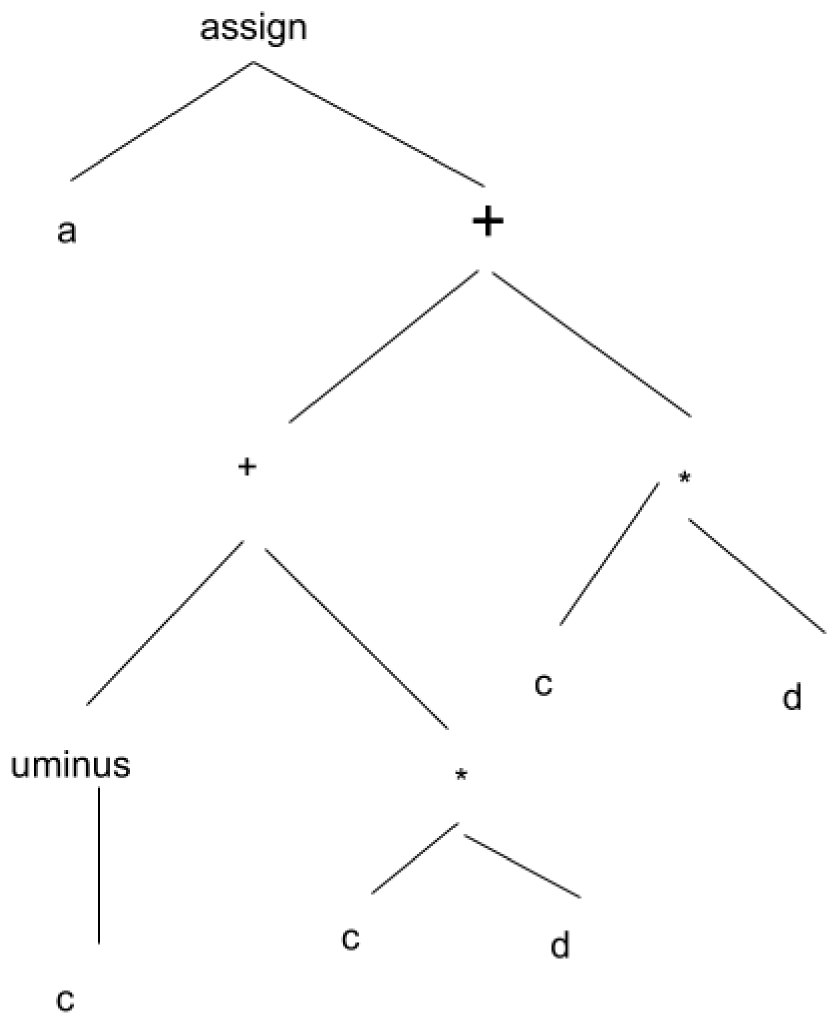
Graphical
Graphically oriented
Heavily used in source to source translations
Tend to be large
Ex 1. Trees
2. DAGs

3.) **What are different representations of 3 address code?**

There are 3 different representations of 3 address code
They are as follows
1. Quadruples
2. Triples
3. Indirect Triples

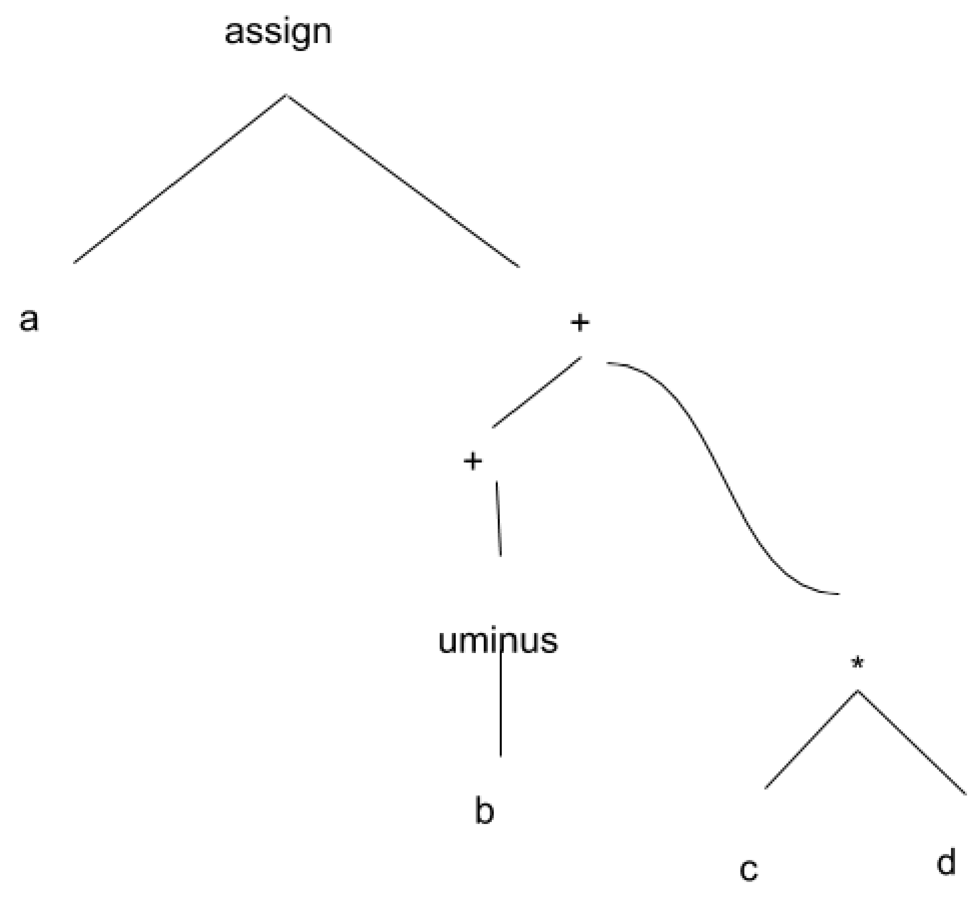
4.) **What is the difference between syntax tree and DAG?**

1. Syntax Trees
They represent constructs in the source program. The children of a node represent the meaningful components
Example



Direct Acyclic Graphs (DAG)

It identifies the common subexpressions and eliminates them.Therefore it is much more efficient as it saves processing time



Conclusion:

We have successfully implemented an Intermediate Code Generator for a code snippet of if conditional construct using Lex and Yacc