Compiler Design Lab 10
Intermediate representations and generating Intermediate representations

Name:- T.V.S.S.Sripad
Roll Number:- 18CS01008

Instructions to Run the codes:-

1) Run the following command :-
   ./run.sh x
   Where x = 1 for AST, x = 2 for 3 Address Code and x = 3 gives DAG

Note:-

1) For AST and DAG generations, after the IR is generated, the output is converted to dot language and then a png image (called “IR.png”) representing the graph is also generated.
2) Lexan1.l and parser1.y contain codes for AST generation.
3) Similarly lexan2.l, parser2,y contain code for 3address code and lexan3.l, parser3.y contain codes for DAG generation.
4) All outputs are written to output.txt file

Q1) Generating AST :-
The Semantic Grammar implemented for generating AST is:-

```
start : E  { printf("Successfully built the AST\n");
             printf("Root is :- %d\n", $1->nodenumber);
             printAST($1); return  0 ; }


E   : E  PLUS T   { $$ = addNode((int)('+'), $1, $3) ;   }
    | E  MINUS  T { $$ = addNode((int)('-'), $1, $3) ;  }
    | T  {$$ = $1; }
    ;

T   : T MULT S { $$ = addNode((int)('*'), $1, $3) ; }
    | T MOD S { $$ = addNode((int)('%'), $1, $3) ; }
    | T DIV S { $$ = addNode((int)('/'), $1, $3) ; }
    | S {$$ = $1; }
    ;

S   : NUM { $$ = $1; }
    | OPEN_BRACKET E CLOSE_BRACKET { $$ = $2; }
    ;
```

Explanation:- In this grammar, $$ stores the pointer to AST that has been generated till that production rule. All production rules of form A -> A op B are associated with a single action i.e., $$ = addNode(). The addNode function takes operator, left, right pointers as arguments and returns a pointer to a newly created node which is then assigned to $$.
The structure of node is given below:-

```
typedef struct Node{
    struct Node *left,*right;
    int val;
    int nodenumber;
    int isOp;
}Node;
```

Each node contains a left pointer, right pointer and a nodenumber (unique to all nodes).
isOp is a variable used to store if a node is an integer node or an Operator node. Finally the val field stores the value of that node. For operator nodes, val stores ASCII values of operator.

Sample Execution:-

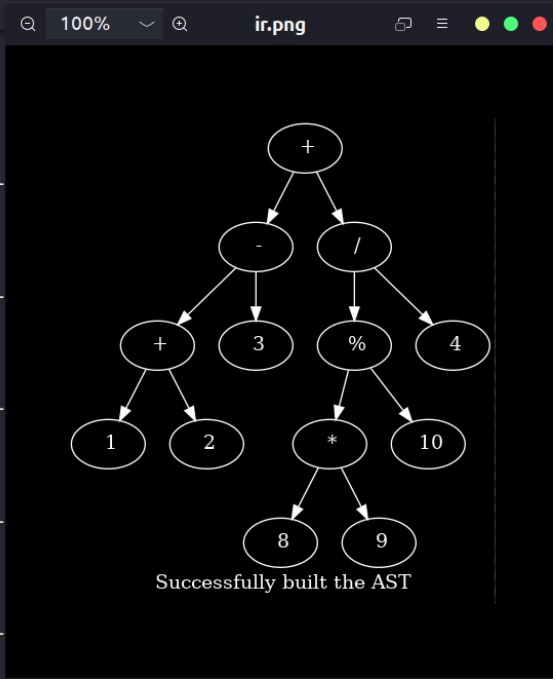
```
sripad cd lab/lab10/18CS01008_LAB10 via 🐍 v3.7.4
➜ ./run.sh 1
1+2-3+(8*9%10/4)

sripad cd lab/lab10/18CS01008_LAB10 via 🐍 v3.7.4
➜
```

Output for the above execution along with generated AST:-

output.txt | lexan1.l | parser1.y

18CS01008_LAB10 > output.txt

```
Successfully built the AST
Root is :- 12
 Node Number :- 0
 Node Value is :- 1
  Left Child and Right Child are :- -1 -1
-----------------------------------------------
 Node Number :- 2
 Node Value is :- +
 Left Child and Right Child are :- 0 1
-----------------------------------------------
 Node Number :- 1
 Node Value is :- 2
  Left Child and Right Child are :- -1 -1
-----------------------------------------------
 Node Number :- 4
 Node Value is :- -
 Left Child and Right Child are :- 2 3
-----------------------------------------------
 Node Number :- 3
 Node Value is :- 3
  Left Child and Right Child are :- -1 -1
-----------------------------------------------
 Node Number :- 12
```



Q2) Generating 3Address Code:-
Grammar implemented for generating 3 address code is:-

```
start : E { printf("Successfully generated 3AC\n"); print3AC($1.code); return 0; }

E   : E PLUS T { sprintf($$.addr, "t%d", variableCounter++); gen($1,$3,&$$,'+'); }
    | E MINUS T { sprintf($$.addr, "t%d", variableCounter++); gen($1,$3,&$$,'-');  }
    | T { strcpy($$.addr, $1.addr); strcpy($$.code, $1.code); }
    ;

T   : T MULT S { sprintf($$.addr, "t%d", variableCounter++); gen($1,$3,&$$,'*');}
    | T DIV S { sprintf($$.addr, "t%d", variableCounter++); gen($1,$3,&$$,'/'); }
    | T MOD S{sprintf($$.addr, "t%d", variableCounter++);   gen($1,$3,&$$,'%'); }
    | S { strcpy($$.addr, $1.addr); strcpy($$.code, $1.code); }
    ;

S   : NUM { sprintf($$.code, "%s", ""); sprintf($$.addr, "%d", $1); }
    | OPEN_BRACKET E CLOSE_BRACKET { strcpy($$.addr, $2.addr); strcpy($$.code, $2.code); }
    ;
```

Explanation:-

Here $$ is a synthesised attribute which has two fields:- code and addr.
All the production rules of form A -> A op B are followed by semantic actions which are:
1) Generation of temporary variable and
2) Generation of 3address code itself.

In the above implementation, the variables are named in the form of t1, t2 and so on. After the temporary variable is created, gen() function is called. The gen() function takes the code and address of the rhs (of production rule) and then appends the instruction corresponding to the production rule to the code which is generated till now and stores the entire code in $$.code attribute.

Similarly for the production:- S -> NUM, the action is $$.addr = the number itself and $$.code = ‘’ (empty).

Sample Execution:-

```
sripad cd lab/lab10/18CS01008_LAB10 via 🐍 v3.7.4
➜ ./run.sh 2
1+2*3-(8*9%10)/4
```

Output for the above input:-

parser2.y | lexan2.l | output.txt

18CS01008_LAB10 > output.txt

```
Successfully generated 3AC
t1 = 2 * 3
t2 = 1 + t1
t3 = 8 * 9
t4 = t3 % 10
t5 = t4 / 4
t6 = t2 - t5
```

Q3) Generation of DAG:-

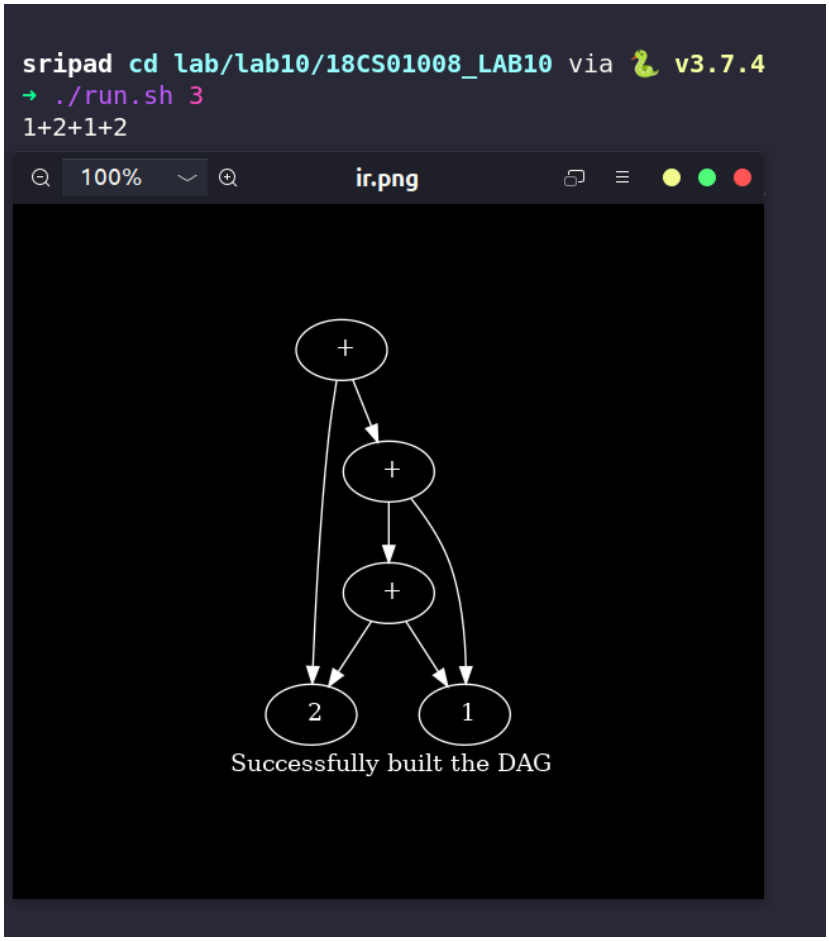
Grammar for generating DAG:-

```
start : E  { printf("Successfully built the DAG\n");
             printf("Root is :- %d\n", $1->nodenumber);
             printDAG($1); return  0 ;
             }


E    : E  PLUS T   { $$ = addNode((int)('+'), 1, $1, $3) ;   }
     | E  MINUS  T { $$ = addNode((int)('-'), 1, $1, $3) ;  }
     | T  {$$ = $1; }
     ;

T    : T MULT S { $$ = addNode((int)('*'), 1, $1, $3) ; }
     | T MOD S { $$ = addNode((int)('%'), 1, $1, $3) ; }
     | T DIV S { $$ = addNode((int)('/'), 1, $1, $3) ; }
     | S {$$ = $1; }
     ;

S    : NUM { $$ = addNode($1,0,NULL,NULL); }
     | OPEN_BRACKET E CLOSE_BRACKET { $$ = $2; }
     ;
```

\

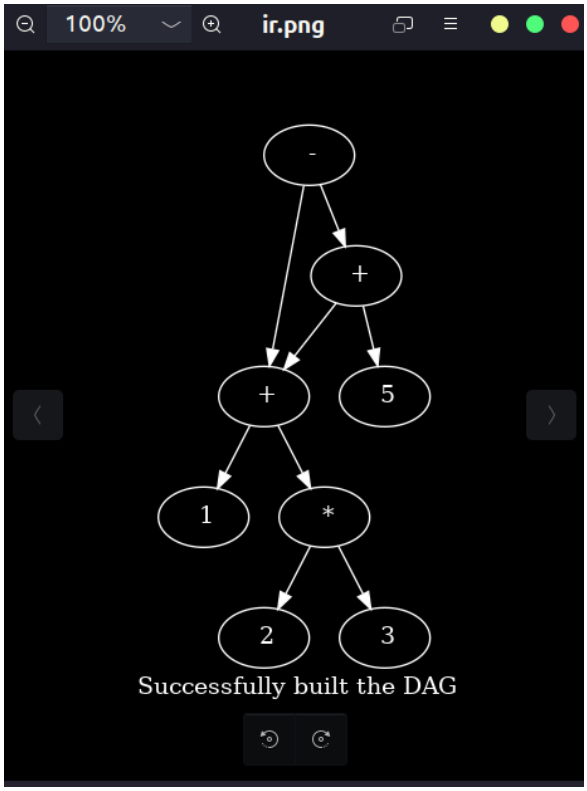
Explanation:-
$$ is a synthesized attribute which stores the pointer to DAG that is generated.
The grammar for generating DAG is very similar to that of AST except that in the addNode() function, before adding a new node, we check if the node is already created or not. To achieve this, all the nodes are stored in the form of a table. Before adding any new node, we check if a node with the same values already exists or not. If such a node already exists in the table then we return the pointer to that node directly without creating any more new nodes. However if it does not exist, we create a new node and add it to the table.

Sample Execution and Outputs:-
Example1:-



Example2:-

Output:-



Q4) GCC intermediate codes:-

The sample codes and intermediate representations generated by gcc are present in gcc-ir folder.

Example1:-

Small sample C-program (fileName:- 1.c) to see 3-address code generated by GCC:-

```c
#include <stdio.h>

int main()
{
    int y = (1 + 2 * 3);
    int x = (1 + 2 * 3) - (5 + y * x) % 12;
    printf("Value is %d\n", x);
    return 0;
}
```

To observe the 3 address code generated by GCC, the file 1.c was compiled with -fdump-tree-gimple flag as follows:-

```
sripad lab10/18CS01008_LAB10/gcc-ir
➜ gcc 1.c -fdump-tree-gimple

sripad lab10/18CS01008_LAB10/gcc-ir
➜ ./a.out
Value is 2
```

The 3 address code generated by GCC was:- (File Name is :- 1.c.005t.gimple)

```
main ()
{
  int D.2318;

  {
    int y;
    int x;

    y = 7;
    _1 = y * x;
    _2 = _1 + 5;
    _3 = _2 % 12;
    x = 7 - _3;
    printf ("Value is %d\n", x);
    D.2318 = 0;
    return D.2318;
  }
  D.2318 = 0;
  return D.2318;
}
```

Sample C program(File Name 2.c) to explore about Control Flow Graph generated by GCC:- (Simple for loop in C)

```c
#include <stdio.h>

int main()
{
    int y = 0;
    int x = 10, i;
    for (i = 0; i < x; i++)
    {
        y++;
    }
    printf("Value is %d\n", y);
    return 0;
}
```

The CFG generated by GCC is:- (obtained using the command **gcc 2.c -fdump-tree-cfg**)

Screenshot of cfg from 2.c.012t.cfg

```
main ()
{
  int i;
  int x;
  int y;
  int D.2322;

  <bb 2> :
  y = 0;
  x = 10;
  i = 0;
  goto <bb 4>; [INV]

  <bb 3> :
  y = y + 1;
  i = i + 1;

  <bb 4> :
  if (i < x)
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]

  <bb 5> :
  printf ("Value is %d\n", y);
  D.2322 = 0;

  <bb 6> :
<L3>:
  return D.2322;
}
```

Explanation of CFG:-

All the initialization of variables is done in the part in the bb2 block.

Then we see a goto statement which states goto bb4.

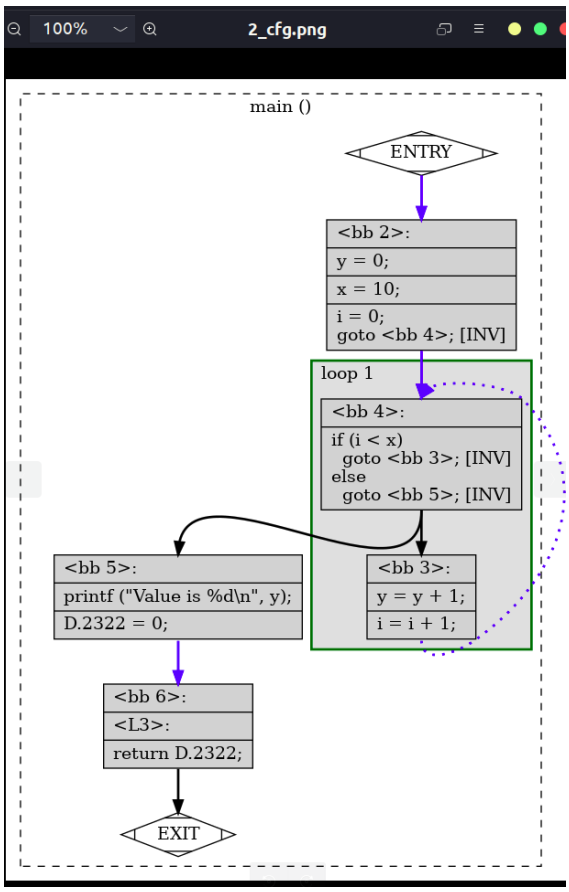
In bb4, we see a if-else statement where if the condition is true we encounter goto bb 3 statement where in bb 3 block, the body of for loop is executed and again we come to bb 4 block where condition of loop is again checked.

If condition doesn;t hold, we encounter goto bb5 statement and bb5 contains the statements to be executed after exiting the for loop

I also found that using the -fdump-tree-cfg-graph flag, we can directly generate dot file for the control flow graph which can then be converted into PNG (or any other image format).
Sample Example:-

```
sripad lab10/18CS01008_LAB10/gcc-ir
➜ gcc -fdump-tree-cfg-graph 2.c

sripad lab10/18CS01008_LAB10/gcc-ir
➜ dot -Tpng 2.c.012t.cfg.dot > 2_cfg.png

sripad lab10/18CS01008_LAB10/gcc-ir
➜
```

The graphic form of CFG generated by GCC for 2.c file is:- (Name:- 2_cfg.png)



Also the ASTs generated for 1.c and 2.c using the -fdump-tree-original-raw flag are included in 1.c.004t.original file and 2.c.004t.original file respectively.