CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

EXPERIMENT – 9(A)

AIM (9A): IMPLEMENTATION OF CODE OPTIMIZATION TECHNIQUES any three (preferable Common sub expression elimination, constant folding, variable propagation)

THEORY:

- Introduction to code optimization

Ans: The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result.

Compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.

The optimization process should not delay the overall compiling process.

- Type of Code optimization techniques

Ans: Common SubExpression Elimination -
The expression that has been already computed before and appears again in the code for computation is called Common SubExpression Elimination.
In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Code Before Optimization	Code After Optimization

S1 = 4 x i S2 = a[S1] S3 = 4 x j S4 = 4 x i // Redundant Expression S5 = n S6 = b[S4] + S5	S1 = 4 x i S2 = a[S1] S3 = 4 x j S5 = n S6 = b[S1] + S5

Constant Folding -
In this technique,
- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

Example-
Circumference of Circle = (22/7) x Diameter
Here,
- This technique evaluates the expression 22/7 at compile time.
- The expression is then replaced with its result 3.14.
- This saves time at run time.

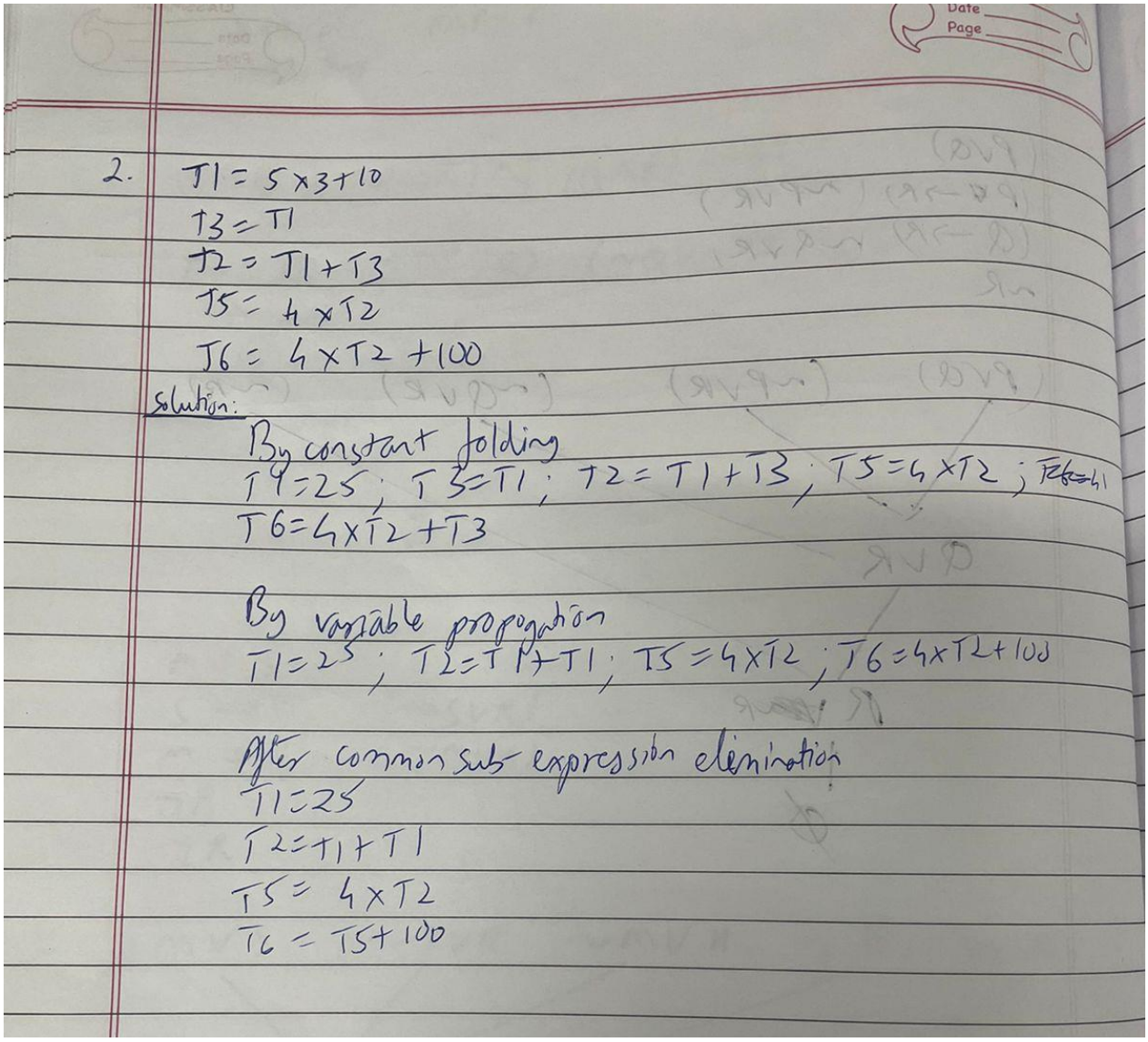
Variable Propagation -
In this technique,
- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of the variable must not get altered in between.

Example-
pi = 3.14
radius = 10
Area of circle = pi x radius x radius
Here,
- This technique substitutes the value of variables 'pi' and 'radius' at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves time at run time.

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

- Working Example

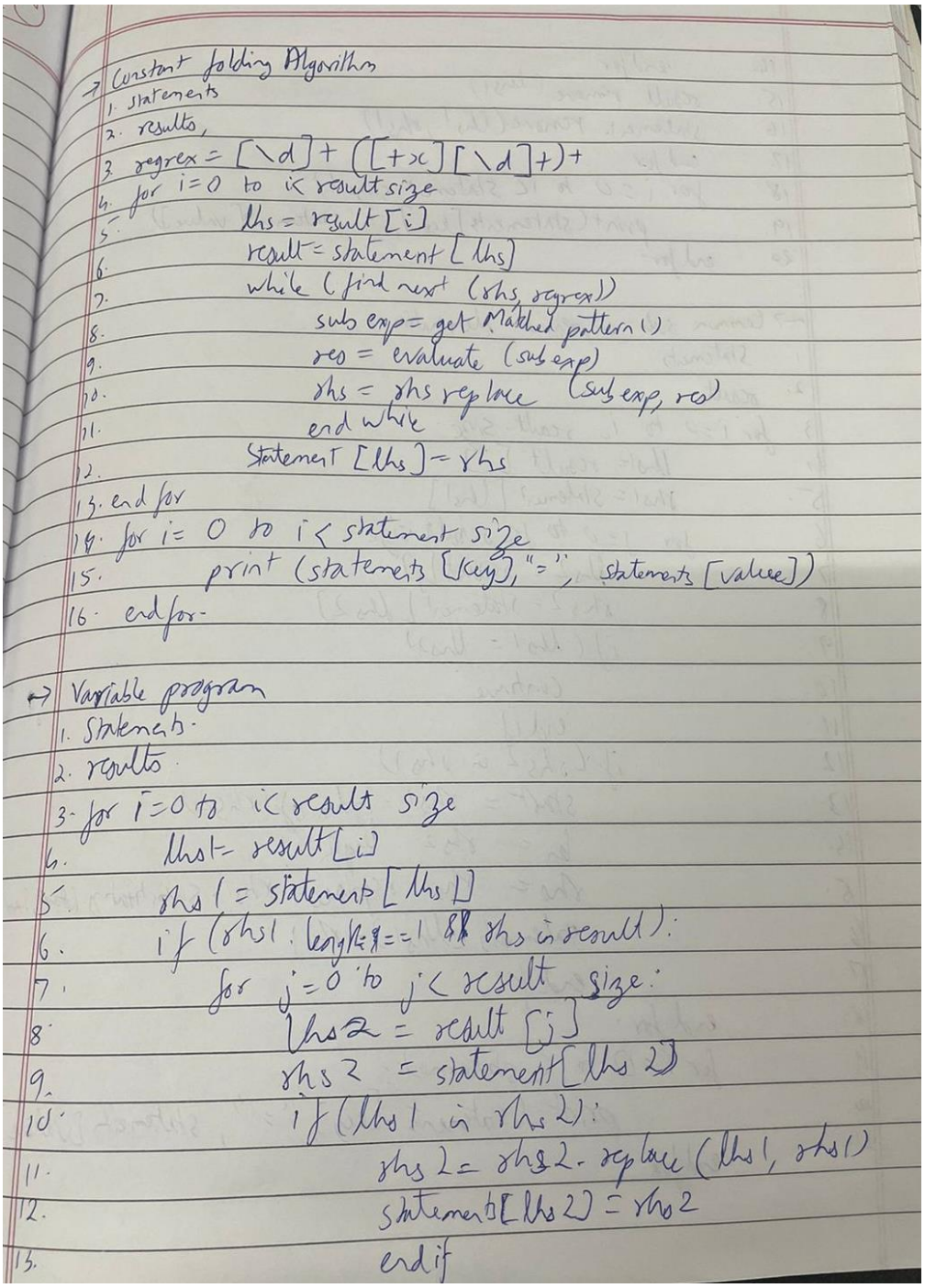
Ans:



CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

IMPLEMENTATION:

- Pseudo code (handwritten Algorithm)



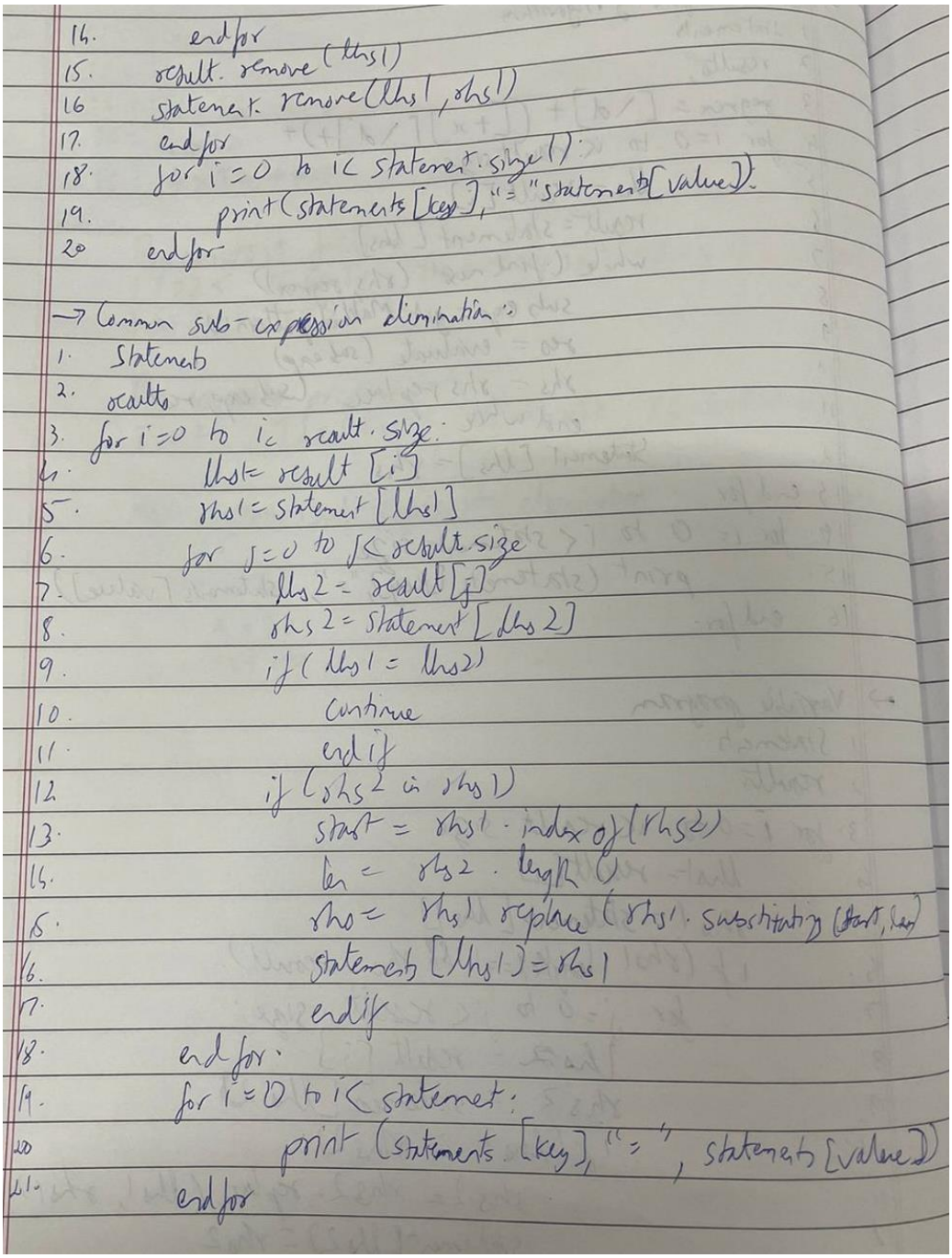
→ Constant folding Algorithm

```
1. statements
2. results,
3. regrex = [\d]+ ([+x][\d]+)+
4. for i=0 to i< result size
5.         lhs = result[i]
6.         result = statement[lhs]
7.         while (find next (rhs, regrex))
8.             sub exp = get Matched pattern()
9.             res = evaluate (sub exp)
10.            rhs = rhs replace (sub exp, res)
11.        end while
12.        Statement[lhs] = rhs
13. end for
14. for i = 0 to i < statement size
15.        print (statements[key], "=", statements[value])
16. end for
```

→ Variable program

```
1. Statements.
2. results
3. for i=0 to i< result size
4.      lhs1 = result[i]
5.      rhs1 = statements[lhs1]
6.      if (rhs1.length()==1 && rhs is result):
7.          for j=0 to j< result size:
8.              lhs2 = result[j]
9.              rhs2 = statement[lhs2]
10.             if (lhs1 in rhs2):
11.                 rhs2 = rhs2.replace(lhs1, rhs1)
12.                 statements[lhs2] = rhs2
13.             endif
```

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28



```
14.     endfor
15.   result.remove(lhs1)
16    statement.remove(lhs1, rhs1)
17.   endfor
18.   for i=0 to i< statement.size():
19.       print(statements[key], "=" statements[value])
20    endfor
```

→ Common sub-expression elimination:

```
1.  Statements
2.  results
3.  for i=0 to i< result.size:
4.      lhs1= result[i]
5.      rhs1 = statement[lhs1]
6.      for j=0 to j< result.size
7.          lhs2 = result[j]
8.          rhs2 = statement[lhs2]
9.          if(lhs1 = lhs2)
10.             continue
11.         endif
12.         if(rhs2 in rhs1)
13.             start = rhs1.indexof(rhs2)
14.             len = rhs2.length()
15.             rhs = rhs1.replace(rhs1.substring(start, len)
16.             statements[lhs1] = rhs1
17.         endif
18.     endfor.
19.     for i=0 to i< statement:
20          print(statements[key], "=", statements[value])
21.     endfor
```

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

- CODE (C / Java) – printed code

```
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class CodeOptimize
{
   HashMap < String, String > statements = new HashMap <> ();
   List < String > result = new ArrayList < String > (Arrays.asList ("a", "b", "c", "d",
"e"));
   List < String > operators =
      new ArrayList < String > (Arrays.asList ("+", "*"));
   public static void main (String[]args)
   {
      CodeOptimize obj = new CodeOptimize();
      obj.getStatements ();
      System.out.println ("Initaially statements are: ");
      obj.putStatements ();
      obj.constantFolding ();
      System.out.println ("After constant folding: ");
      obj.putStatements ();
      obj.variablePropagation ();
      System.out.println ("After variable propagation: ");
      obj.putStatements ();
      obj.commonSubexpElim ();
      System.out.println ("After Common Sub-expression Elimination: ");
      obj.putStatements ();
   }
   public void getStatements ()
   {
      this.statements.put ("a", "5*3+10");
      this.statements.put ("b", "a");
      this.statements.put ("c", "a+b");
      this.statements.put ("d", "4*c");
      this.statements.put ("e", "4*c+100");
   }
   public void putStatements ()
   {
      for (Map.Entry mapElement:this.statements.entrySet ())
      {
         String key = (String) mapElement.getKey ();
         String value = (String) mapElement.getValue ();
         System.out.println (key + " : " + value);
      }
      System.out.println ("----------------------------------------");
   }
```

```
public int evaluate (String str)
{
   String[]arr = str.split ("\\+");
   for (int i = 0; i < arr.length; i++)
   {
      int result = 1;
      if (arr[i].contains ("*"))
      {
         String[]num = arr[i].split ("\\*");
         for (int j = 0; j < num.length; j++)
         {
            result *= Integer.parseInt (num[j]);
         }
         arr[i] = String.valueOf (result);
      }
   }
   int len = arr.length;
   int sum = 0;
   for(int i = 0; i < len; i++)
   {
      sum += Integer.parseInt (arr[i]);
   }
   return sum;
}
public void constantFolding ()
{
   for (int i = 0; i < this.result.size (); i++)
   {
      String lhs = this.result.get (i);
      String rhs = this.statements.get (lhs);
      Pattern p = Pattern.compile ("[\\d]+([+*][\\d]+)+");
      Matcher m = p.matcher (rhs);
      while (m.find ())
      {
         String subexpr = m.group ();
         int result = this.evaluate (subexpr);
         String res = String.valueOf (result);
         rhs = rhs.replace (rhs.substring (m.start (), m.end ()), res);
         m = p.matcher (rhs);
      }
      this.statements.put (lhs, rhs);
   }
}
public void variablePropagation ()
{
```

```
    for (int i = 0; i < this.result.size (); i++)
    {
       String lhs1 = this.result.get (i);
       String rhs1 = this.statements.get (lhs1);
       if (rhs1.length () == 1 && this.result.contains (rhs1))
       {
          for (int j = 0; j < this.result.size (); j++)
          {
             String lhs2 = this.result.get (j);
             String rhs2 = this.statements.get (lhs2);
             if (rhs2.contains (lhs1))
             {
                rhs2 = rhs2.replace (lhs1, rhs1);
                this.statements.put (lhs2, rhs2);
             }
          }
          this.result.remove (lhs1);
          this.statements.remove (lhs1, rhs1);
       }
    }
}
public void commonSubexpElim ()
{
    for (int i = 0; i < this.result.size (); i++)
    {
       String lhs1 = this.result.get (i);
       String rhs1 = this.statements.get (lhs1);
       for (int j = 0; j < this.result.size (); j++)
       {
          String lhs2 = this.result.get (j);
          String rhs2 = this.statements.get (lhs2);
          if (lhs1 == lhs2)
          {
             continue;
          }
          if (rhs1.contains (rhs2))
          {
             int start = rhs1.indexOf (rhs2);
             int len = rhs2.length ();
             rhs1 = rhs1.replace(rhs1.substring(start,len),lhs2);
             this.statements.put(lhs1,rhs1);
          }
       }
    }
}
```

}

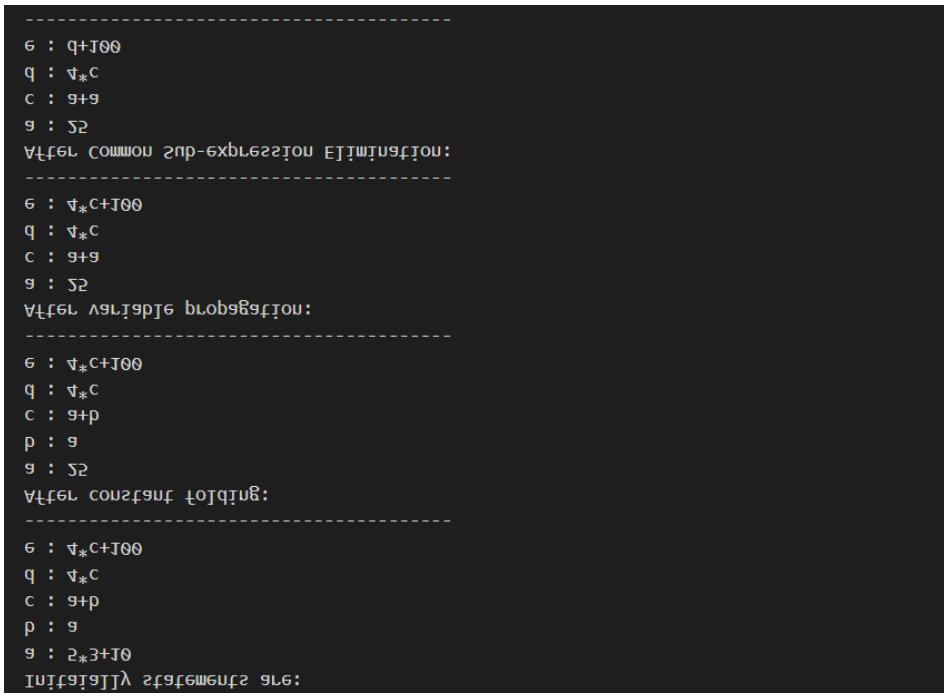
OUTPUT:

Program and output to optimize a given code.
Input program (Three address code) to be given as a text file.

Input.txt (before optimization)
T1= 5*3+10 // Constant folding
T3=T1 //variable propagation
T2=T1+T3
T5=4*T2 // common sub-expression elimination T6=4*T2+100

Output (After Optimization)

T1=25
T2=T1+T1
T5=4*T2
T6=T5+100



CONCLUSION: From the above experiment we are able to learn about various code optimization techniques like the constant folding, variable propagation and common subexpression elimination.

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

Aim 9 (B): TO IMPLEMENT INTERMEDIATE CODE GENERATION ALGORITHM.

Theory:

- What is the role of Code Generation in Compiler Design?

Ans: Code Generation -

Code generation can be considered as the final phase of compilation. Through post code generation, optimization processes can be applied on the code, but that can be seen as a part of the code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

- What are the issues in design of Code generation?

Ans: The following issue arises during the code generation phase:

1. Input to code generator –
   The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc.
2. Target program –
   The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
3. Memory Management –
   Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.
4. Instruction selection –
   Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered.

5. Evaluation order –
   The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.
6. Approaches to code generation issues: Code generators must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:
   - Correct
   - Easily maintainable
   - Testable
   - Efficient

- Explain Code Generation Algorithm with the help of an example.

Ans: Code Generation Algorithm -
The algorithm takes a sequence of three-address statements as input. For each three address statement of the form a:= b op c perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y is currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z has no next uses or does not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those registers will no longer contain y or z.

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

Example -

1. t:= a-b
2. u:= a-c
3. v:= t +u
4. d:= v+u

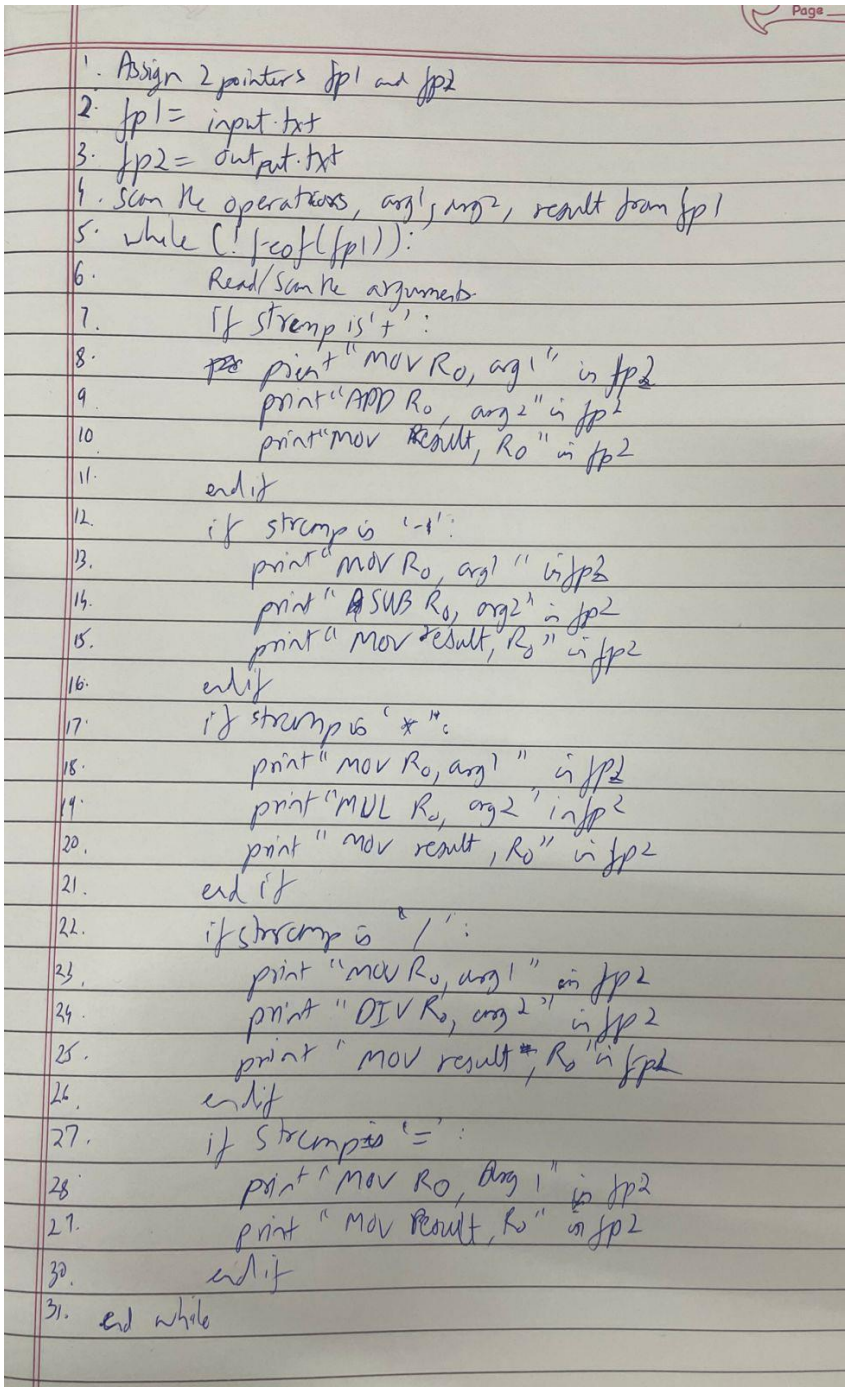
Code Sequence is as follows -

Statement	Code Generated
t:= a-b	MOV a, R0 SUB b, R0
u:= a-c	MOV a, R1 SUB c, R1
v:= t +u	ADD R1, R0
d:= v+u	ADD R1, R0 MOV R0, d

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

IMPLEMENTATION:

- Pseudo code (handwritten Algorithm)



1. Assign 2 pointers fp1 and fp2
2. fp1 = input.txt
3. fp2 = output.txt
4. Scan the operators, arg1, arg2, result from fp1
5. while (!feof(fp1)):
6. Read/Scan the arguments
7. if strcmp is '+':
8. print "MOV R0, arg1" in fp2
9. print "ADD R0, arg2" in fp2
10. print "mov Result, R0" in fp2
11. endif
12. if strcmp is '-':
13. print "MOV R0, arg1" in fp2
14. print "SUB R0, arg2" in fp2
15. print "MOV Result, R0" in fp2
16. endif
17. if strcmp is '*':
18. print "MOV R0, arg1" in fp2
19. print "MUL R0, arg2" in fp2
20. print "MOV result, R0" in fp2
21. end if
22. if strcmp is '/':
23. print "MOV R0, arg1" in fp2
24. print "DIV R0, arg2" in fp2
25. print "MOV result, R0" in fp2
26. endif
27. if Strcmp is '=':
28. print "MOV R0, Arg1" in fp2
29. print "MOV Result, R0" in fp2
30. endif
31. end while

- CODE (C / Java) – printed code of each

codegeneration.c -

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char op[2],arg1[5],arg2[5],result[5];
int main()
{
  FILE *fp1,*fp2;
  fp1=fopen("input.txt","r");
  fp2=fopen("output.txt","w");
  while(!feof(fp1))
  {


    fscanf(fp1,"%s%s%s%s",op,arg1,arg2,result);
    if(strcmp(op,"+")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nADD R0,%s",arg2);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
    if(strcmp(op,"*")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nMUL R0,%s",arg2);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
    if(strcmp(op,"-")==0)
    {
      fprintf(fp2,"\nMOV R0,%s",arg1);
      fprintf(fp2,"\nSUB R0,%s",arg2);
      fprintf(fp2,"\nMOV %s,R0",result);
    }
```

```
  if(strcmp(op,"/")==0)
  {
   fprintf(fp2,"\nMOV R0,%s",arg1);
   fprintf(fp2,"\nDIV R0,%s",arg2);
   fprintf(fp2,"\nMOV %s,R0",result);
  }
      if(strcmp(op,"=")==0)
  {
   fprintf(fp2,"\nMOV R0,%s",arg1);
   fprintf(fp2,"\nMOV %s,R0",result);
  }
 }
 fclose(fp1);
 fclose(fp2);
 return 0;
}
```

input.txt -

```
- a b t
- a c u
+ t u v
+ v u d
```

OUTPUT:

Input :

3AC for the statement → d : = (a-b) + (a-c) + (a-c)

```
t = a - b
u = a - c
v = t + u
d = v + u
```

CLASS: TE CMPN B DATE: 19/03/ '22 NAME: Keegan Vaz ROLL NO.: 28

Statements	Code Generated	Register descriptor Register empty	Address descriptor
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0

d : = v + u	ADD R1, R0
	
	MOV R0, d
		R0 contains d	d in R0
			d in R0 and memory

output.txt -

```
MOV R0,a
SUB R0,b
MOV t,R0
MOV R0,a
SUB R0,c
MOV u,R0
MOV R0,t
ADD R0,u
MOV v,R0
MOV R0,v
ADD R0,u
MOV d,R0
MOV R0,v
ADD R0,u
MOV d,R0
```

CONCLUSION:

The machine code is generated for 3AC using a code generation algorithm.