EXPERIMENT 10

CLASS: TE CMPN A　　　　　　　　　　ROLL NO. : 19
NAME: REBECCA DIAS　　　　　　　　　PID: 182027

AIM: TO IMPLEMENT CODE GENERATION PHASE OF COMPILER.

THEORY:

What is the role of Code Generation in Compiler Design?

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?

The following issue arises during the code generation phase:

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. The code generation phase just proceeds on an assumption that the inputs are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

Target program –

The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.

Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.

Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loaders. But there is added expense of linking and loading.

Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

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.

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 plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

P:=Q+R

```
S:=P+T

MOV Q, R0

ADD R, R0

MOV R0, P

MOV P, R0

ADD T, R0

MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

Register allocation issues –

Use of registers makes the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:

During Register allocation – we select only those set of variables that will reside in the registers at each point in the program.

During a subsequent Register assignment phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machines require register pairs consisting of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

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.

Approaches to code generation issues: Code generator 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

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each three-address statement of the form

x : = y op z, perform the following actions:

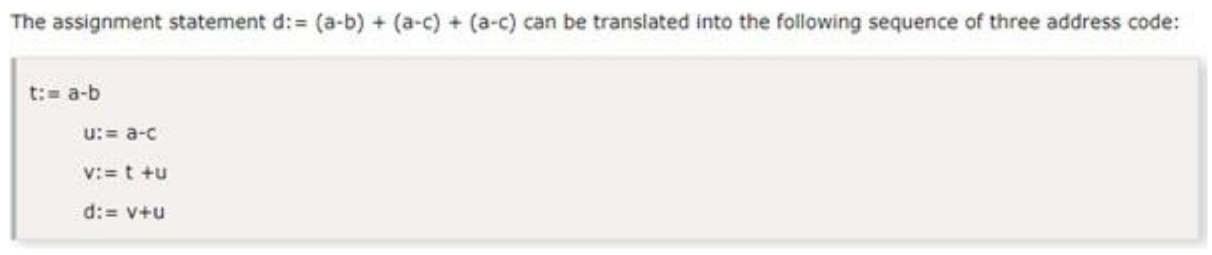
Step 1 : Invoke a function getreg() to determine the location L where the result of the computation y op z should be stored.

Step 2 : Determine the present location of ‘y’ by consulting the address descriptor of y. If y is not present in the location ‘L’ then generate the instruction MOV y’, L to copy value of y to L

Step 3 : The present Location of Z is determined using step 2 and the instruction is generated as OP z’ , L

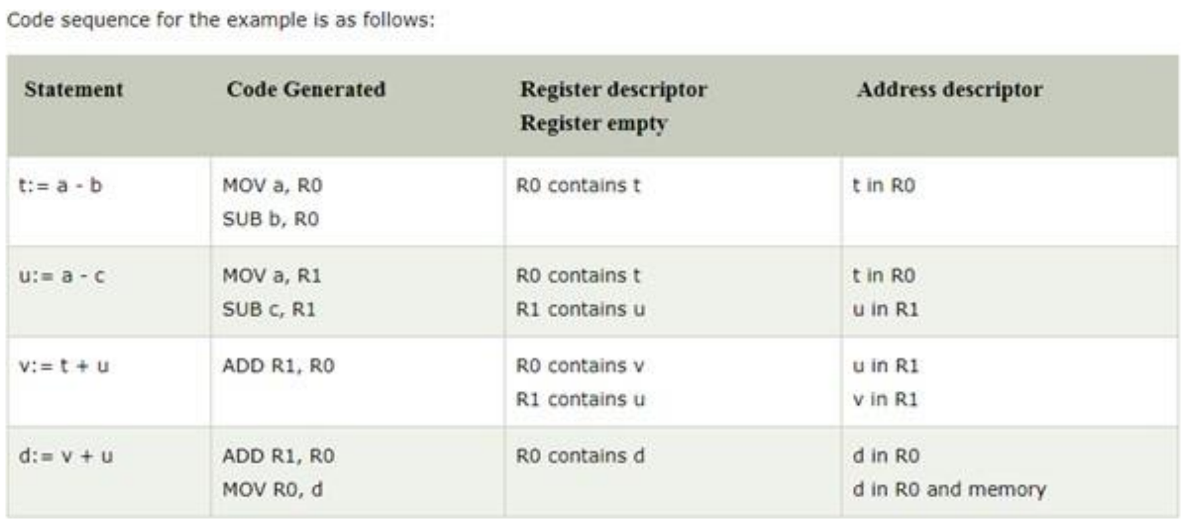
Step 4 : Now L contains the value of y op z i.e. assigned to x. So, if L is a register then update its descriptor that contains the value of x. Update address Descriptor of x to indicate that it is stored in ‘L’.

Step 5 : if y, z have no future use then update the Descriptor to remove y and Z



The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

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

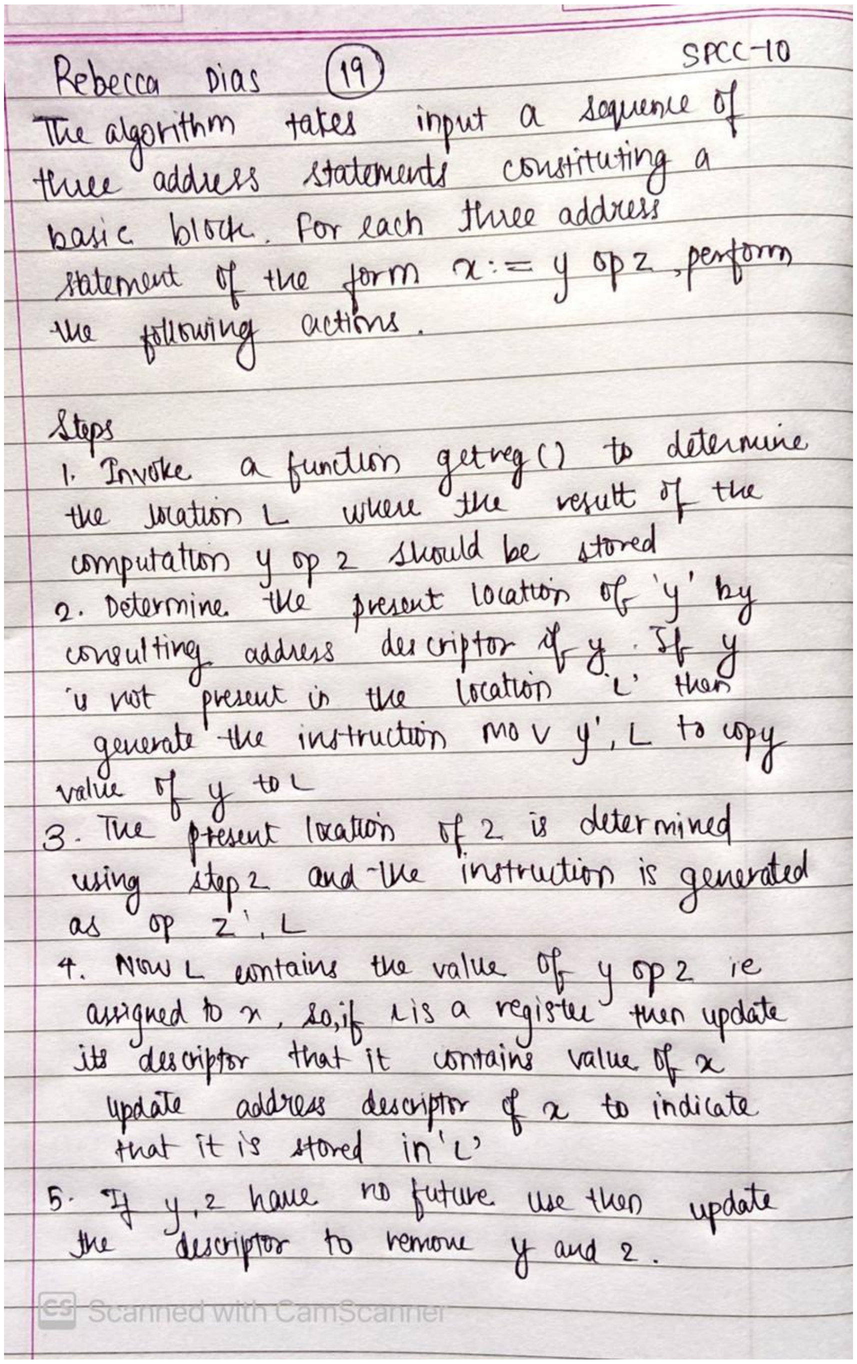


Code sequence for the example is as follows:

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

Implementation:

Pseudo Code:



Rebecca Dias (19) SPCC-10

The algorithm takes input a sequence of three address statements constituting a basic block. For each three address statement of the form $x := y \text{ op } z$, perform the following actions.

Steps

1. Invoke a function getreg() to determine the location L where the result of the computation y op z should be stored
2. Determine the present location of 'y' by consulting address descriptor of y. If y is not present in the location 'L' then generate the instruction MOV y', L to copy value of y to L
3. The present location of z is determined using step 2 and the instruction is generated as op z', L
4. Now L contains the value of y op z ie assigned to x, so, if L is a register then update its descriptor that it contains value of x update address descriptor of x to indicate that it is stored in 'L'
5. If y, z have no future use then update the descriptor to remove y and z.

Code:

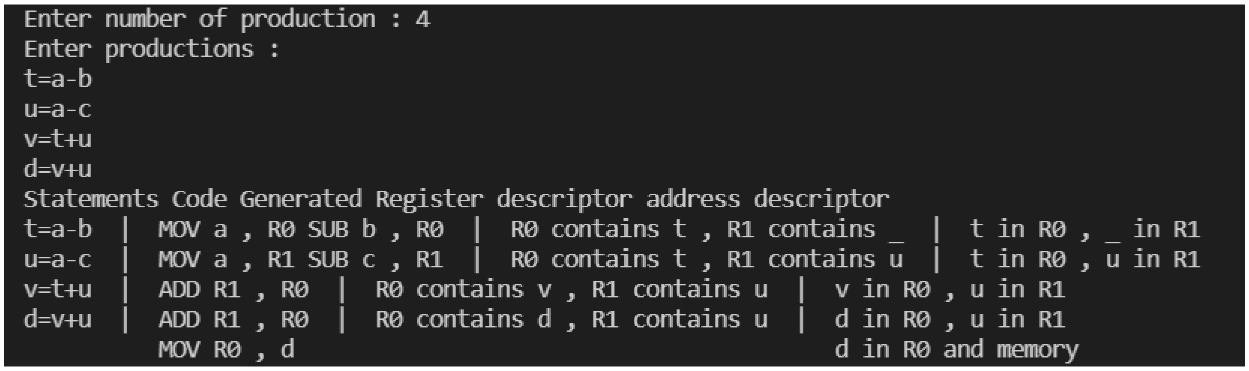
```
import random
t=int(input("Enter number of production : "))
print("Enter productions  : ")
d={}
R0=""
R1=""
left={}
count=0
code_generated=[]
register_descriptor=[]
address_descriptor=[]
statements=[]
for i in range(t):
    s=input()
    statements.append(s)
    q=list(s.split("="))
    d[str(q[0])]=str(q[1])
for each in d.keys():
    l=[]

    c=list(tuple(d[each]))
    if len(R0)==0:
        l.append("MOV")
        l.append(c[0])
        l.append(",")
        l.append("R0")
        if c[1] =="+":
            l.append("ADD")
        if c[1] =="-":
            l.append("SUB")
        l.append(c[2])
        l.append(",")
        l.append("R0")
        R0=each
    elif len(R1)==0:
        l.append("MOV")
        l.append(c[0])
        l.append(",")
        l.append("R1")
        if c[1] =="+":
            l.append("ADD")
        if c[1] =="-":
            l.append("SUB")
        l.append(c[2])
        l.append(",")
        l.append("R1")
        R1=each
    elif c[0]==R0:
        if c[1]=="+":
            l.append("ADD")
        if c[1]=="-":
            l.append("SUB")
        l.append("R1")
        l.append(",")
        l.append("R0")
        R0=each
    code_generated.append(l)
    q=["R0","contains",R0,",","R1","contains",R1 if R1!="" else "_"]
    register_descriptor.append(q)
    q1=[R0, "in", "R0",",",R1 if R1!="" else "_","in","R1"]
    address_descriptor.append(q1)

l=["MOV","R0",",","d"]
code_generated.append(l)

print("Statements Code Generated Register descriptor address descriptor")
for i in range(4):
    print(statements[i]," | ",*code_generated[i]," |
",*register_descriptor[i]," | ",*address_descriptor[i])
final=code_generated[-1][0]+"  "+code_generated[-1][1]+"
"+code_generated[-1][2]+" "+code_generated[-1][3]
print("\t ",final,end=" ")
print("\t\t\t\t\t  ",end="")
print(*address_descriptor[-1][:3],end="  ")
print("and memory")
```

Output:



```
Enter number of production : 4
Enter productions :
t=a-b
u=a-c
v=t+u
d=v+u
Statements Code Generated Register descriptor address descriptor
t=a-b  |  MOV a , R0 SUB b , R0  |  R0 contains t , R1 contains _  |  t in R0 , _ in R1
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  |  v in R0 , u in R1
d=v+u  |  ADD R1 , R0  |  R0 contains d , R1 contains u  |  d in R0 , u in R1
          MOV R0 , d                                        d in R0 and memory
```

Conclusion:

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