Name: Bhavesh Kewalramani
Roll No.: A-25

Practical No. 10

Theory

Code Generator

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Example:

Consider the three address statement x:= y + z. It can have the following sequence of codes:

MOV x, R_0
ADD y, R_0

Register and Address Descriptors:

- A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.
- An address descriptor is used to store the location where current value of the name can be found at run time.

A 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 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 descriptor.

4. If the current value of y or z have no next uses or 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 register will no longer contain y or z.

Generating Code for Assignment Statements:

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

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

Practicals

Aim:

Write a program to generate the code using simple code generation algorithm.

Program:

```
input1 = ["a=b+c","d=e+f","t=g+a","r=d+t"]
# input1 = ["t=a+b","u=c+d",v=t-u","x=v+u"]

registers = 2
registerStatus = [0, 0]
memory = []
memory_dict = {}


def CodeGenerator(input1):
    print("Instructions Generated...")
    print()
    for i in input1:
        ops = [i[0], i[2], i[4]]
        operator = i[3]
        con1 = True
        con = False
        for reg in registerStatus:
            if reg == ops[1]:
                con1 = False
        if con1:
            for j in range(registers):
                if registerStatus[j] == 0:
                    con = True
                    registerStatus[j] = ops[1]
                    print("MOVE", ops[1], ",", "R" + str(j))
                    break
            if not con:
                oc1 = registerStatus[0]
                oc2 = registerStatus[1]
                flag2 = -1
                found1 = False
                found2 = False
                for eqn in input1[input1.index(i):]:
                    a, b = eqn.split("=")
                    op1 = b[0]
                    op2 = b[2]
                    if (op1 == oc1 or op2 == oc1) and not found1:
                        found1 = True
                        flag2 = 0
                        flag1 = True
                    if (op1 == oc2 or op2 == oc2) and not found2:
                        found2 = True
                        flag2 = 1
                        flag1 = True
```

```
        if not found2:
            flag1 = False
            flag2 = 1
        if not found1:
            flag1 = False
            flag2 = 0

        if flag2 == 0:
            if flag1:
                print("\nMoving R0 to memory....")
                memory.append(registerStatus[0])
                memory_dict[registerStatus[0]] = memory.index(registerStatus[0])
                print("Transfered to memory.....")
            print("R0 is now empty\n")
            if ops[1] in memory:
                print("MOVE", "MEM" + str(memory_dict[ops[1]]), ",", "R0")
            else:
                print("MOVE", ops[1], ",", "R0")
            registerStatus[0] = ops[1]
        else:
            if flag1:
                print("\nMoving R1 to memory....")
                memory.append(registerStatus[1])
                memory_dict[registerStatus[1]] = memory.index(registerStatus[1])
                print("Transfered to memory.....")
            print("R1 is now empty\n")
            if ops[1] in memory:
                print("MOVE", "MEM" + str(memory_dict[ops[1]]), ",", "R1")
            else:
                print("MOVE", ops[1], ",", "R1")
            registerStatus[1] = ops[1]

    for reg in range(len(registerStatus)):
        if registerStatus[reg] == ops[1]:
            if operator == '+':
                if ops[2] in registerStatus:
                    a = registerStatus.index(ops[2])
                    print("ADD", "R" + str(a), ",", "R" + str(reg))
                else:
                    print("ADD", ops[2], ",", "R" + str(reg))
            if operator == '-':
                if ops[2] in registerStatus:
                    a = registerStatus.index(ops[2])
                    print("SUB", "R" + str(a), ",", "R" + str(reg))
                else:
```

```
                print("SUB", ops[2], ",", "R" + str(reg))
            registerStatus[reg] = ops[0]
        print("Status : ", registerStatus)
```

CodeGenerator(input1)

Output:

```
Instructions Generated...

MOVE b , R0
ADD c , R0
Status :  ['a', 0]
MOVE e , R1
ADD f , R1
Status :  ['a', 'd']

Moving R1 to memory....
Transfered to memory......
R1 is now empty

MOVE g , R1
ADD R0 , R1
Status :  ['a', 't']
R0 is now empty

MOVE MEM0 , R0
ADD R1 , R0
Status :  ['r', 't']
```

```
Instructions Generated...

MOVE a , R0
ADD b , R0
Status :  ['t', 0]
MOVE c , R1
ADD d , R1
Status :  ['t', 'u']
SUB R1 , R0
Status :  ['v', 'u']
ADD R1 , R0
Status :  ['x', 'u']
```

```
In [ ]:
```