1 2 3 4 5 6 8

The left of 3 is 4. As parent of 4 is 3 and that is listed hence list 4. Left of 4 is 5 which has listed parent (i.e. 4) hence list 5. Similarly list 6.

1 2 3 4 5 6 8

As now only 8 is remaining from the unlisted interior nodes we will list it.

Hence the resulting list is 1 2 3 4 5 6 8. Then the order of computation is decided by reversing this list. We get the order of evaluation as 8 6 5 4 3 2 1. That also means that we have to perform the computations at these nodes in the given order.

$t_8 := d/e$

$t_6 := a - b$

$t_5 := t_6 + c$

$t_4 := t_5 * t_8$

$t_3 := t_4 - e$

$t_2 := t_6 + t_4$

$t_1 := t_2 * t_3$

This gives the optimized code for DAG even though there are any number of registers.

8.12.3 Labelling Algorithm

The labelling algorithm generates the optimal code for given expression in which minimum registers are required. Using labelling algorithm the labelling can be done to the tree by visiting nodes in bottom-up order. By this all the child nodes will be labelled before its parent nodes.

For computing the label at node n with the label L1 to left child and label L2 to the right child as

$$\text{Label}(n) = \begin{cases} \max(L1, L2) & \text{if } L1 \neq L2 \\ L1 + 1 & \text{if } L1 \neq L2 \end{cases}$$

- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.
- If labels of the children of a node n are L1 and L2 respectively then

$$\begin{aligned} \text{Label}(n) &= \begin{cases} \max(L1, L2) & \text{if } L1 \neq L2 \\ L1 + 1 & \text{if } L1 = L2 \end{cases} \end{aligned}$$

Example 8.5 : *Label the following tree*

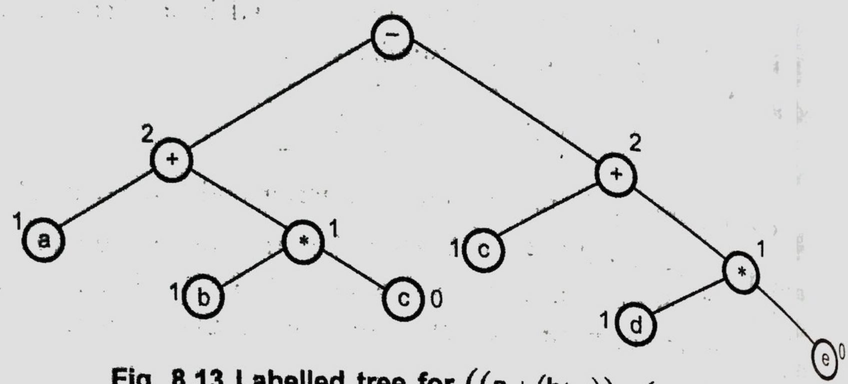

Fig. 8.13 Labelled tree for $((a+(b*c))-(c+(d*e)))$

Solution :

Algorithm

We will first discuss the important notations used in this algorithm.

1. The function Gen_code(n) is used that produces the code to evaluate a n labelled as n.
2. For the register allocation a stack is used called 'Reg_Stack'. Initially stack contains all the available registers R0, R1, ...Rk such that the register is on the top of the stack.
3. The Gen_code(n) evaluates n in the register which is on the top of the sta
4. Another stack is used to store the temporaries called 'Temp_stack' it cont all the temporaries T0, T1,...Tk having T0 on the top of the stack.
5. The function swap(Reg_Stack) is used to swap top two registers on the st
6. The function Print is used to concatenate the arguments in the order in w they are coming. The || is used to indicate the concatenation. In this func the non-quoted arguments need to be evaluated first.
7. The function Gen_code is a function that generates the code. This func consists of various cases. Let us consider these cases.

Case 1: If n is a left node and it is a leaf node.

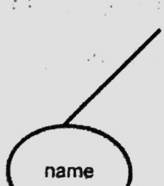

Fig. 8.14

`Print('MOV'||name||',' top(Reg_Stack)|`

Here the node n is named by an ident *name*. This also means load *name* into register.

Case 2: If node's right child is a leaf n_2 then,

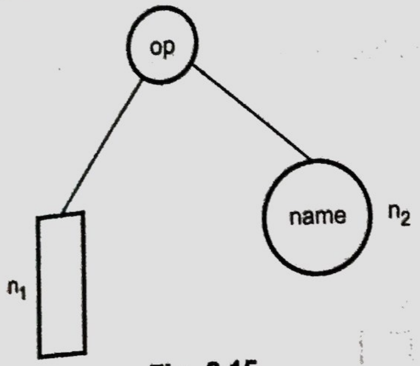

Fig. 8.15

```
Gen_code(n1)
Print(op||name||','||top(Reg_stack));
```

Here n_1 is first evaluated in the register on the top of the stack and then the operation on identifier *name* is performed with the same register which is on the top of the stack.

Case 3:

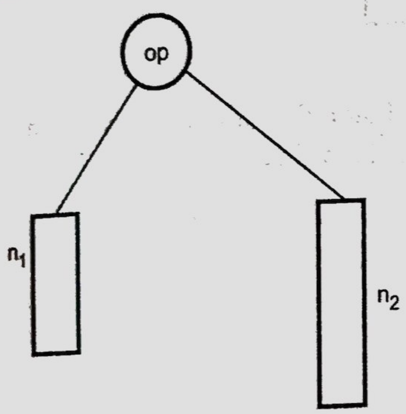

Fig. 8.16

If left child of n i.e. n_1 requires less number of registers than n_2 then we swap top two registers on Reg_Stack.

Then evaluate n_2 into R = top (Reg_Stack). We remove R from the Reg_stack and evaluate n_1 into then we generate an instruction

```
Print(op||R||','||top(Reg_Stack);
```

Then we push R onto the Reg_stack and then call for swap.

Thus in this case we evaluate the right subtree into register and store this register just below the top of the stack. Then we evaluate the left subtree into the register which is on the top of the stack. The register on the top of the stack contains the left operand and register below the top of the stack contains right operand. Then n is evaluated in the register on the top of the stack.

Case 4 : If the left child n_1 requires more number of registers than the right child n_2, then,

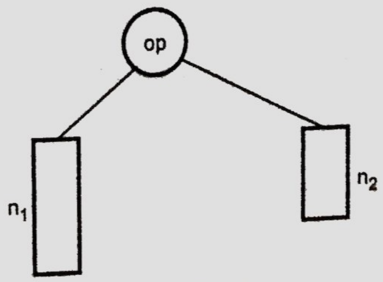

Fig. 8.17

In this case we evaluate left subtree first then the right subtree. There is no need to swap the registers here.

Case 5: Both the children require equal or more number of registers than the available number of registers. Then,

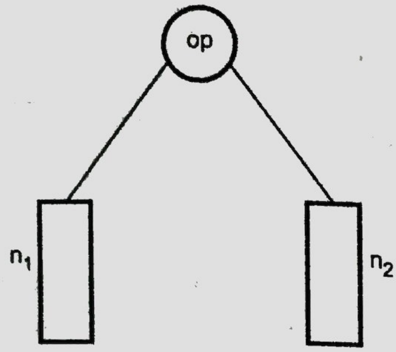

Fig. 8.18

In this case the right subtree is evaluated first and stored in a temporary. Then the evaluation of left subtree is done. And finally the root is evaluated.

Code generation from labelled tree

The function Gen_code(n) can be written as

```
Gen_code(n)
{
/*  case 1: */
if n is a left leaf node then
  Print('MOV'||name||','||top(Reg_Stack);
else
if n is an interior node with operator op and left, right child as
n1 and n2 respectively then
/* case 2 */
if Label(n2)=0
{
n2 represents the operand name
Gen_code(n1);
Print(op||name||','||top[Reg_Stack);
}
/*case 3 */
else if(Label(n1)< Label(n2) AND Label(n1) < k
{
swap(Reg_Stack);
Gen_code(n1);
R=pop(Reg_Stack)   /* n2 is evaluated in reg.R*/
Gen_code(n1);
Print(op||R||','||top(Reg_Stack);
Push(Reg_Stack,R);
swap(Reg_Stack);
```

```
}
/* Case 4 */
else if(Label(n2)< Label(n1) AND Label(n2) < k
  /* k is total number of registers*/
{
Gen_code(n1);
R=pop(Reg_Stack)   /* n1 is evaluated in reg.R*/
Gen_code(n2);
Print(op||R||','||top(Reg_Stack);
Push(Reg_Stack,R);
}
/* case 5*/
else
 {
  Gen_code(n2);
  T:=pop(Temp_Stack);
  Print('MOV'||top(Reg_Stack)||','||T);
  Gen_code(n1);
  Push(Temp_Stack,T);
  Print(op||T||','||top(Reg_Stack));
 }
}
```

Let us generate a code from this labelling algorithm.

Consider the labelled tree as follows.

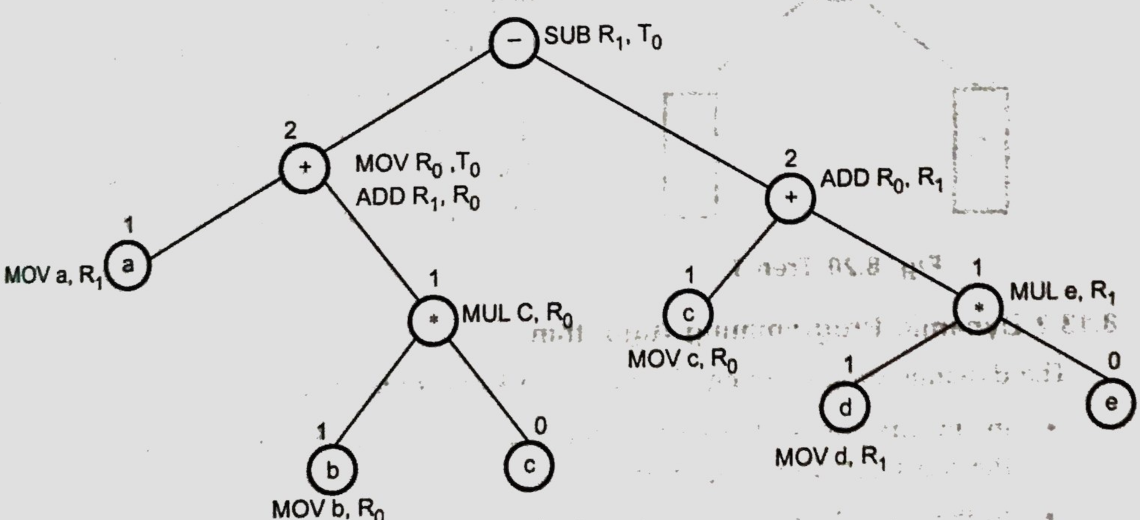

Fig. 8.19 Labelled tree with generated code for $((a+(b*c))-(c+(d*e)))$

```
MOV b, R0
MUL c, R0
MOV a, R1
ADD R1, R0
MOV R0, T0
```

```
MOV d, R1
MUL e, R1
MOV c, R0
ADD R0, R1
SUB R1, T0
```