DAG Construction, Reordering and labeling

Directed Acyclic Graph - DAG

- Useful data structures for implementing transformation on basic blocks
- Gives detail of how the value is computed for each statement of the basic block

DAG - Application

- Common sub-expression
- Which names are inside the block but evaluated outside
- Which statements of a block gets their value computed outside the block

DAG construction

- Leaves are labeled by unique identifiers – variable or constants – represent r-values
- Interior nodes are labeled by an operator
- Nodes are optimally given a sequence of identifiers

DAG construction algorithm

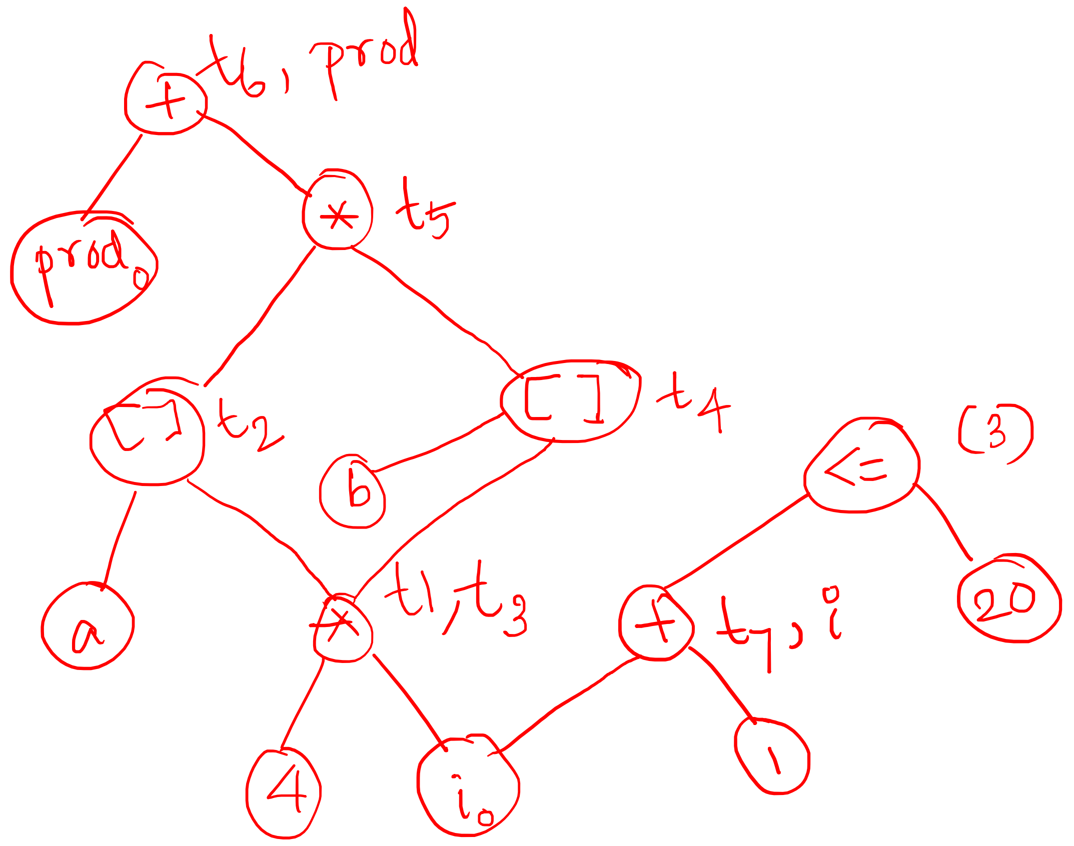
- Input: Basic block
- Output: DAG having label for each node – operator or variables . For each node a list of attached identifiers
- (i) x := y op z
- (ii) x := op y
- (iii) x := y
- If node (y) is undefined create a label ‘y’ and let it be node (y) and so for node (z) if ‘z’ is not there

DAG algorithm

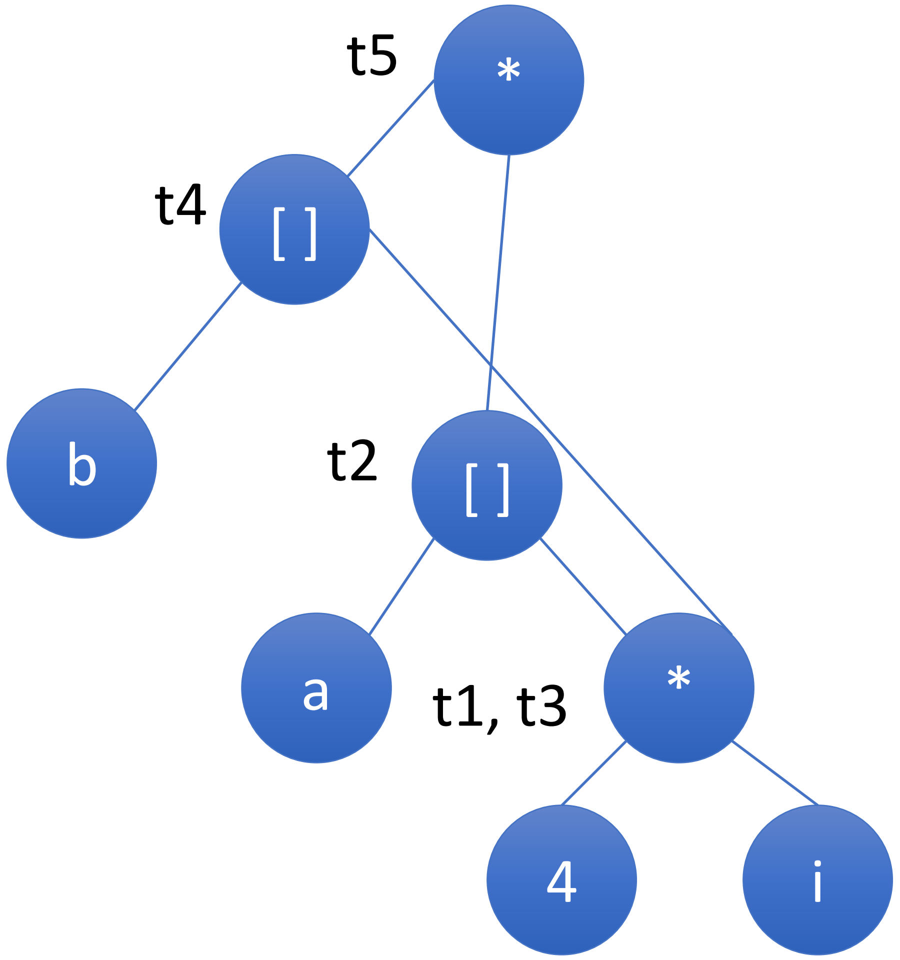
- In case (i) check if there is a node labelled ‘op’ whose left and right children are ‘y’ and ‘z’ respectively. If not create such a node and let it be ‘n’. In case (ii) ignore ‘z’ and do the same thing. In case (iii) let the node(y) be ‘n’
- Delete x from the list of attached identifiers for node(x). Append ‘x’ to the list of attached identifiers for the node ‘n’ and set node(x) to n

Example

```
3.    t1 := 4 * i
4.    t2 := a[t1]
5.    t3 := 4 * i
6.    t4 := b [t3]
7.    t5 := t2 *t4
8.    t6 := prod + t5
9.    prod := t6
10.   t7 := i+1
11.   i := t7
12.   If i <= 20 goto (3)
```



DAG

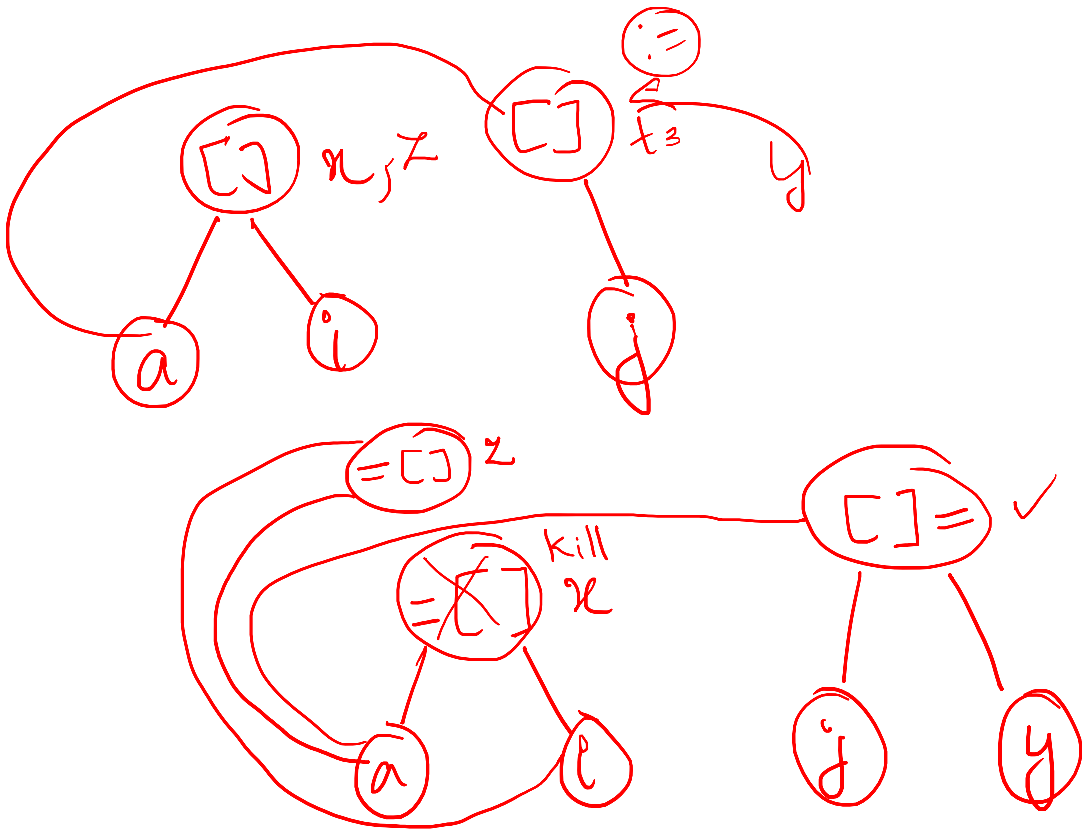


Arrays, Pointers and Procedures

- x := a[i]
- a[j] := y
- z := a[i]

Same as

- x := a[i]
- z := x
- a[j] := y



Arrays, Pointers and Procedures

- If i = j, then the value chosen is different for ‘z’
- So, while processing array, we kill all nodes labeled [] so, that confusion is prevented
- Similar scenarios is adopted for pointers and procedures

Modification to algorithm

- Evaluation of or assignment to an array must follow the previous assignment if there is one
- Any assignment to an array element follow any previous evaluation of a
- Any use of an identifier must follow the previous procedure call or indirectly through a pointer if there is one

Modification to algorithm

- Any procedure call or indirect assignment through a pointer must follow all previous evaluations of an identifier

Code generation from DAG

- Rearranging the order
- Heuristic ordering for DAGs
- Optimal ordering for Trees
  - Labeling the tree for register
  - Tree traversal to generate code

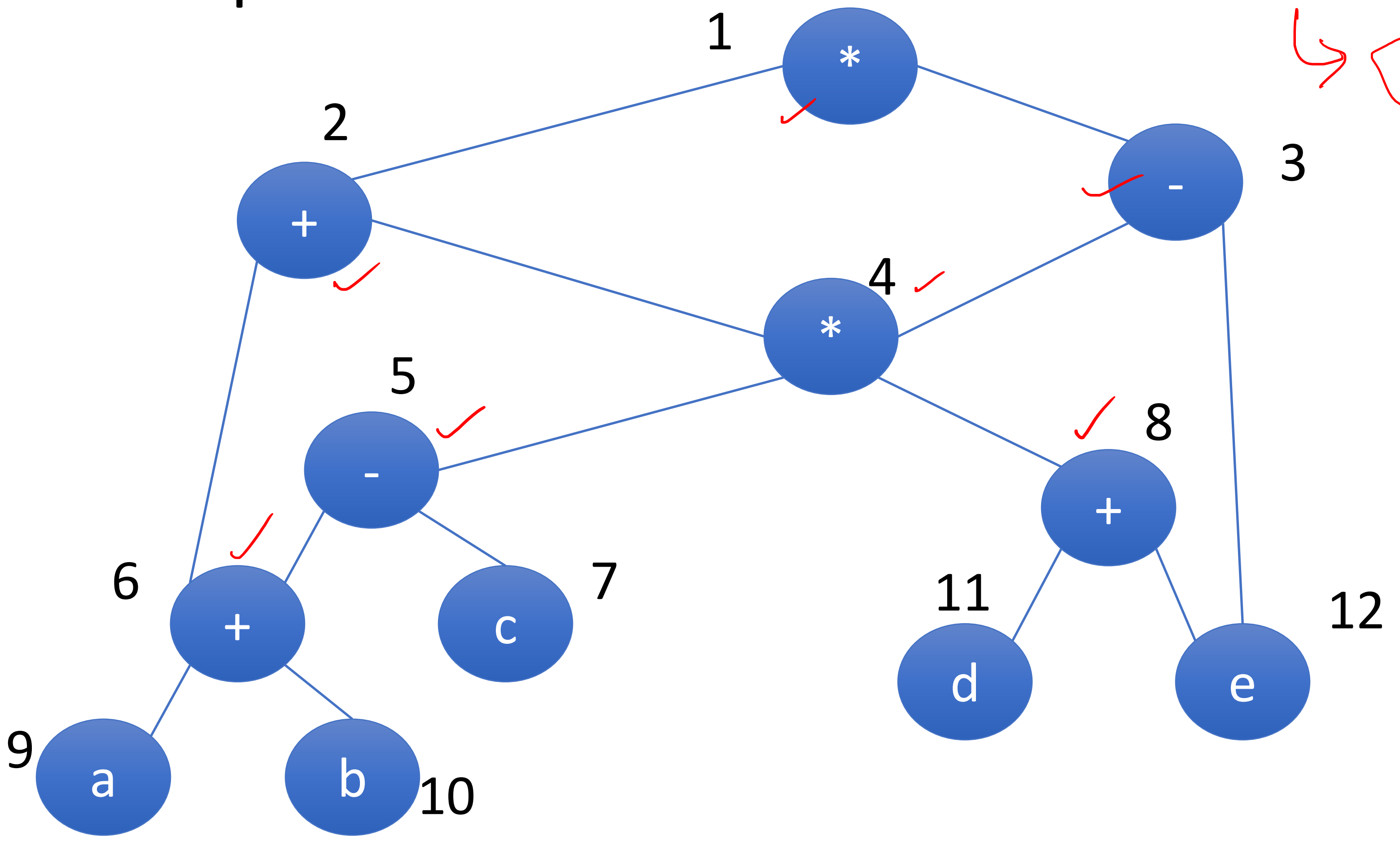
Rearranging the order

- Changing the order of independent statements to efficiently utilize the registers.
- Reduce the final cost of assembly level code

Node listing algorithm

```
while unlisted interior nodes remain do begin
      select an unlisted node n, all of whose parents have been listed ;
  list n;
       while the leftmost child m of n has no unlisted parents
                    and is not a leaf do
                         /* since n was just listed , m is not yet listed*/
                         begin
                         list m;
                         n = m
                     end
    end
```

Example



1 2 3 4 5 6 8

↳ 8 6 5 4 3 2 1

$t_8 = d + e$

$t_6 = a + b$

$t_5 = t_6 - c$

$t_4 = t_5 * t_8$

$t_3 = t_4 - e$

$t_2 = t_6 + t_4$

$t_1 = t_2 * t_3$

Example

- Ordering of the sequence 1234568
- Reversed 8654321
- Evaluate t8 first
- t8 := d +e
- t6 := a +b
- t5 := t6 - c

Example

- t4 := t5 * t8
- t3 := t4 – e
- t2 := t6 + t4
- t1:= t2 + t3

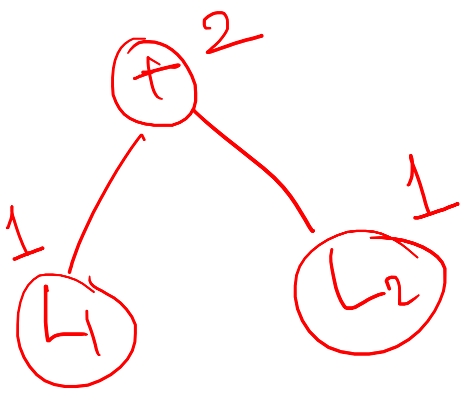
Labeling algorithm

- “left leaf” means a node that is a leaf and the left most descendent of its parent. All other leaves are referred to as “right leaves”.
- Labeling can be be done by visiting node in a bottom up order so that a node is not visited until all its children all labeled.

Labeling algorithm

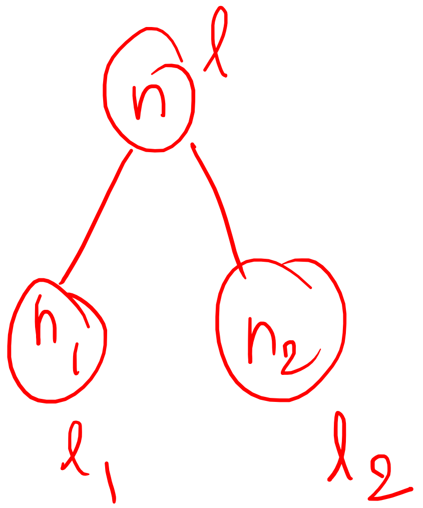
- The order in which the first three nodes are created is suitable if the parse tree is used as intermediate code, so in this case the labels can be computed as syntax directed translation.
- label (n) = max (l1, l2) if l1 != l2

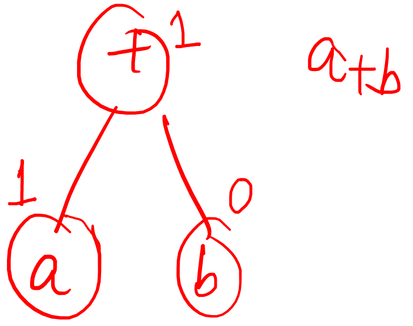
  l1 + 1 if l1 == l2



label(lc) = 1

label (rc) = 0





MOV a, R0

ADD b, R0

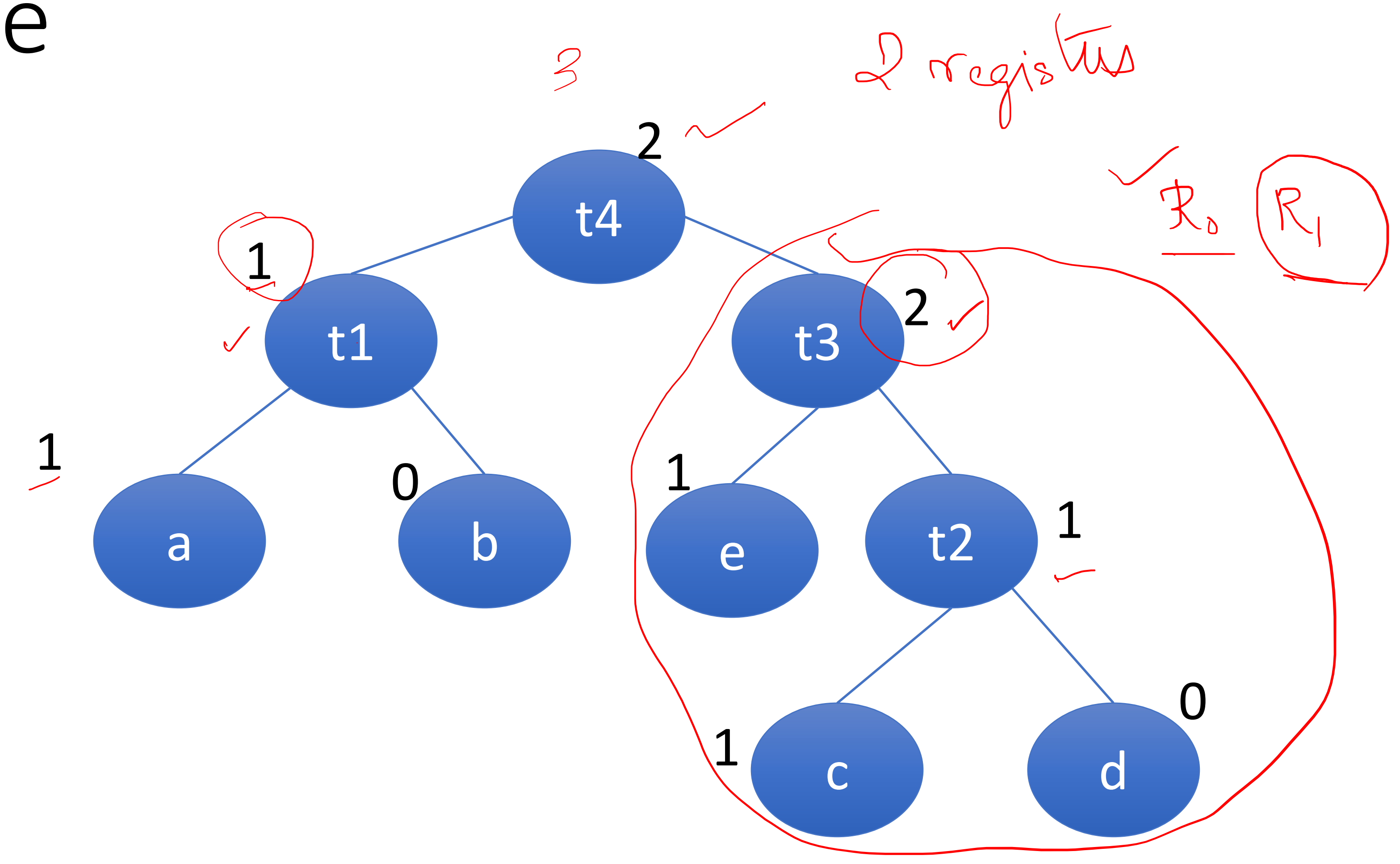
Label computation

- Post order traversal is used
- Node a is labelled 1 since it is the left most leaft node.
- b is labelled 0 as it is the right child
- Parent of a, b is assigned max(1,0) which is 1

Labeling algorithm

```
if n is a leaf then
    if n is leftmost child of its parents  then
        Label (n) = 1
    else label (n) = 0
else begin  /*  n is an interior node */
     let n1, n2 , …. , nk be the children of n ordered by label ,
             so label (n1) >= label (n2) >= ….>= label (nk) ;
      label (n) = max (label(ni) + i - 1)
end
```

Example



Label computation

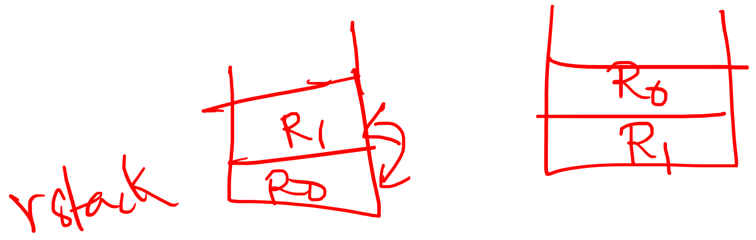
- The labelling is carried out till the root and the value at a node indicates the number of registers required to compute the node

Code generation from DAG

- Uses a recursive procedure on a labelled DAG
- Considers code generation based on the labels assigned to the nodes
- “stack rstack” to allocate registers. Initially rstack contains all available registers.

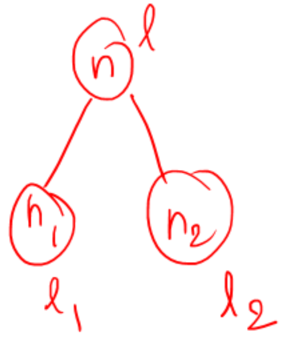
Code generation from DAG

- Leaves the registers on rstack in the same order it found them.
- The function swap(rstack) interchanges the top two registers on rstack.



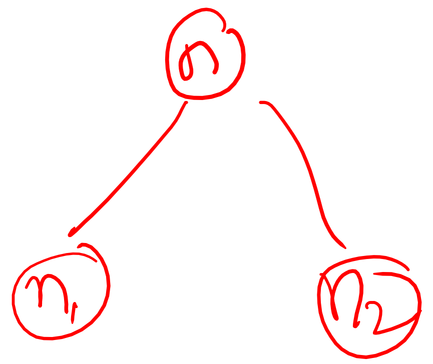
Five cases to generate CODE

- **Case 0:** n is a leaf and the leftmost child of its parent. Therefore, we generate just a load instruction.
- **Case 1:** we have a subtree of the form for which we generate code to evaluate n_1 into register R=top(rstack) followed by the instruction
    - op name, R.



Five cases to generate CODE

- **Case 2:** A subtree of the form where n_1 can be evaluated without stores but n_2 is harder to evaluate than n_1 as it requires more registers. For this case, swap the top two registers on rstack, then evaluate n_2 into R=top(rstack). We remove R from rstack and evaluate n_1 into S = top(rstack). Then generate the instruction op R, S, which produce the value of n in register S. Another call to swap leaves rstack as it was when this call of gencode begins.

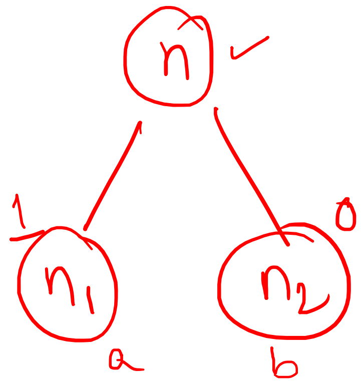


Five cases to generate CODE

- **Case 3**: It is similar to case 2 except that here the left subtree is harder and is evaluated first. There is no need to swap registers here.
- **Case 4:** It occurs when both subtrees requires r or more registers to evaluate without stores. Since we must use a temporary memory location, we first evaluate the right subtree into the temporary T, then the left subtree, and finally the root.

Algorithm

```
Procedure gencode(n);
Begin
/* case 0 */
if n is a left leaf representing operand name and n is the leftmost child of its parent then
        print 'MOV' || name || ',' || top(rstack)
else if n is an interior node with operator op, left child $n_1$, and right child $n_2$ then
/* case 1 */
        if label($n_2$) = 0 then begin
              let name be the operand represented by $n_2$;
             gencode($n_1$);
             print op || name || ',' || top(rstack)
        end
```



Algorithm

```
/* case 2 */
        else if 1 ≤ label (n$_1$) < label(n$_2$) and label(n$_1$) < r then begin
                swap(rstack);
                gencode(n$_2$ );
                R := pop(rstack); /* n$_2$  was evaluated into register R */
                gencode(n$_1$);
                print op || R || ‘,’ || top(rstack);
                push(rstack, R);
                swap(rstack)
         end
```

Algorithm

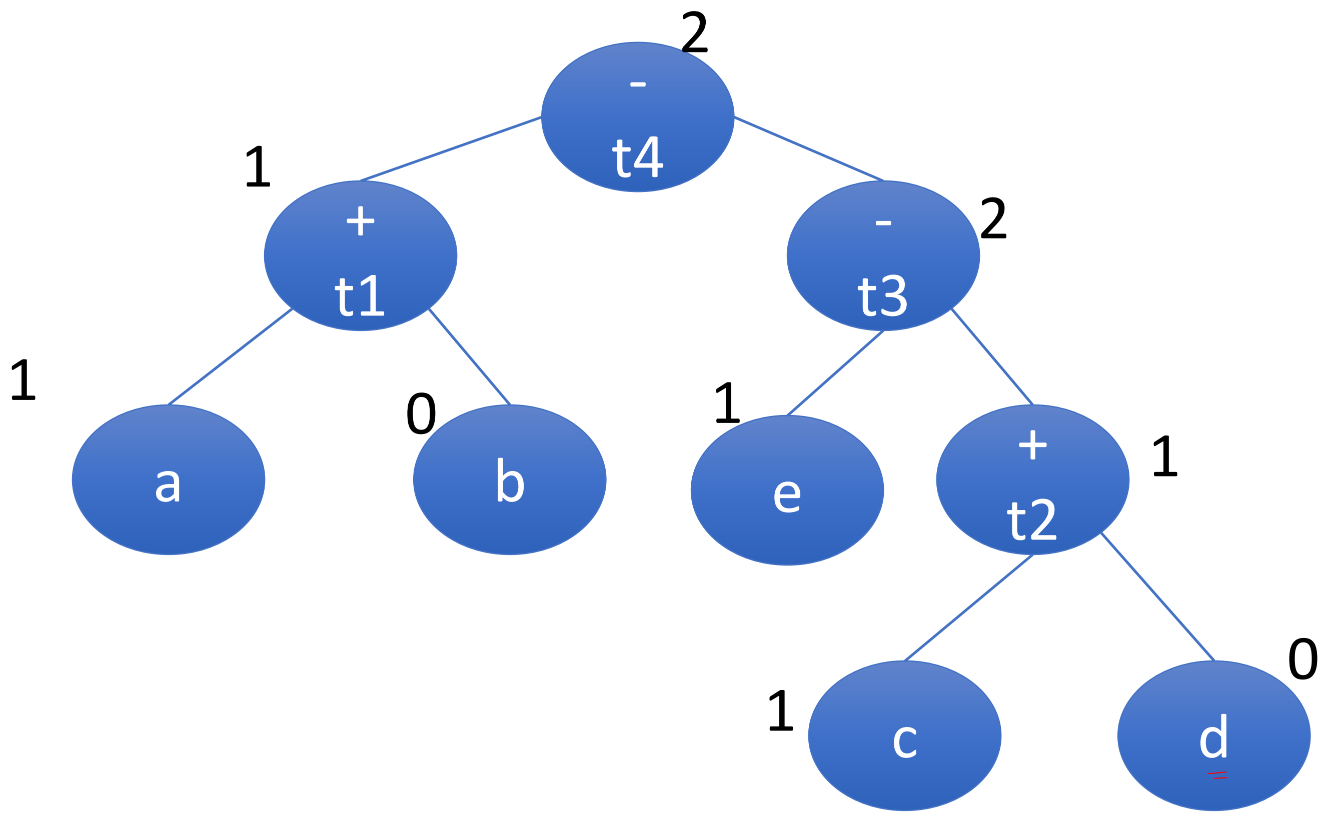
```
/* case 3 */
        else if $1 \leq$ label ($n_2$) $\leq$ label($n_1$) and label($n_2$) < r then begin
            gencode($n_1$);
            R := pop(rstack); /* $n_1$  was evaluated into register R */
            gencode($n_2$);
            print op || top(rstack) || ‘,’ ||R;
            push(rstack, R);
         end
```

Algorithm

```
/* case 4, both labels ≥ r, the total number of registers */
        else begin
                gencode($n_2$ );
                T := pop(tstack);
                Print ‘MOV’ || top(rstack) || ‘,’ || T;
                gencode($n_1$);
                push(tstack, T);
                print op || T || ‘,’ || top(rstack)
       end
end
```

gencode(t4)

Example



Code

```
gencode($t_4$)                              [R1 R0]     // case 2
    gencode($t_3$)                          [R0 R1]     // case 3
        gencode(e)                          [R0 R1]     // case 0
            print MOV e, R1
        gencode($t_2$)                      [R0]        // case 1
            gencode(c)                      [R0]        // case 0
                print MOV c, R0
            print ADD d, R0
        print SUB R0, R1                    [R0 R1]
  gencode($t_1$)                            [R0]        // case 1
      gencode(a)                            [R0]        // case 0
          print MOV a, R0
      print ADD b, R0
  print SUB R1, R0
```

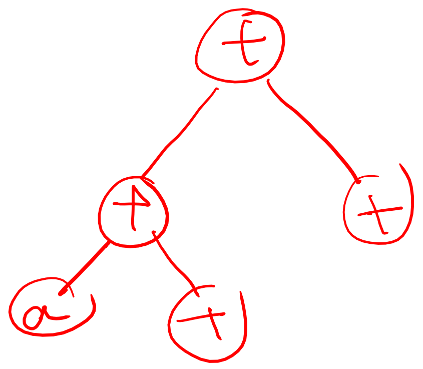
Multi-register operation

- Supports only one register based operation
- To support multi-register the label algorithm is changed

$$\text{label}(n) = \begin{cases} \max(2, l1, l2) & \text{if } l1 \neq l2 \\ l1 + 1 & \text{if } l1 = l2 \end{cases}$$

Algebraic Properties

- Use to swap left and right nodes to effectively use the code generation algorithm
- Common sub-expression need not be recomputed.



Summary

- DAG construction
- DAG in the context of array and pointers
- Heuristic reordering of statements for reducing instruction costs
- Labeling algorithm to identify the number of registers required to compute a node
- Code generation