TITLE: Code generation using DAG / labeled tree.

THEORY:

Directed Acyclic Graph

A directed acyclic graph, is a directed graph with no directed cycles. That is, it is formed by a collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again.

DAGs may be used to model many different kinds of information. The reachability relation in a DAG forms a partial order, and any finite partial order may be represented by a DAG using reachability. A collection of tasks that must be ordered into a sequence, subject to constraints that certain tasks must be performed earlier than others, may be represented as a DAG with a vertex for each task and an edge for each constraint; algorithms for topological ordering may be used to generate a valid sequence. Additionally, DAGs may be used as a space-efficient representation of a collection of sequences with overlapping subsequences. DAGs are also used to represent systems of events or potential events and the causal relationships between them. DAGs may also be used to model processes in which data flows in a consistent direction through a network of processors, or states of a repository in a version-control system.

The corresponding concept for undirected graphs is a forest, an undirected graph without cycles. Choosing an orientation for a forest produces a special kind of directed acyclic graph called a polytree. However there are many other kinds of directed acyclic graph that are not formed by orienting the edges of an undirected acyclic graph. Moreover, every undirected graph has an acyclic orientation, an assignment of a direction for its edges that makes it into a directed acyclic graph. For these reasons it would be more accurate to call directed acyclic graphs acyclic directed graphs or acyclic digraphs.

Use in Compiler Design

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

ALGORITHM

1. For Leaf Nodes
   Assign label 1 to left child node and label 0 to right child node.

2. For Interior Nodes
   Case 1: If Node's children's labels are different, then
   Node's Label = Maximum among Children's Labels.
   Case 2: If Node's children's labels are same, then
   Node's Label = (Left Child's Label OR Right Child's Label) + 1.

Algorithm for Generating Assembly Code:

(Say, R is a Stack consists of registers)

```
void gencode(Node)
{
if Node is intermediate node of tree(DAG)
{
   Case 1: if Node's left child's label == 1 && Node's right child's label == 0
    {
     print "MOV Node's left child's data,R[top]"
     print "op Node's right child's data,R[top]"
    }

  Case 2: else if Node's left child's label >= 1 && Node's right child's label == 0
    {
     gencode(Node's left child);
     print "op Node's right child's data,R[top]"
    }

 Case 3: else if Node's left child's label < Node's right child's label
    {
     int temp;
     Swap Register Stack's top and second top element;
     gencode(Node's right child);
     temp=pop();
     gencode(Node's left child);
     push(temp);
     Swap Register Stack's top and second top element;
     print "op R[top-1],R[top]"
    }

 Case 4: else if Node's left child's label >= Node's right child's label
    {
     int temp;
     gencode(Node's left child);
     temp=pop();
     gencode(Node's right child);
     push(temp);
     print "op R[top-1],R[top]"
    }

}

else if Node is leaf node and it is left child of it's immediate parent
{
 print "MOV Node's data,R[top]"
}

}
```

OUTPUT:

$ g++ DAG.cpp

$ ./a.out

Enter root of tree: +

Enter number of children of +: 2

Enter Left Child of +: -

Enter Right Child of + :/

Enter number of children of -: 2

Enter Left Child of -: a

Enter Right Child of -: *

Enter number of children of a :0

Enter number of children of * :2

Enter Left Child of *: b

Enter Right Child of *: c

Enter number of children of b :0

Enter number of children of c :0

Enter number of children of / :2

Enter Left Child of / :a

Enter Right Child of / :-

Enter number of children of a :0

Enter number of children of - :2

Enter Left Child of - :d

Enter Right Child of - :c

Enter number of children of d :0

Enter number of children of c :0

Inorder Display:
a with Label 1
- with Label 2
b with Label 1
* with Label 1
c with Label 0
+ with Label 3
a with Label 1
/ with Label 2
d with Label 1
- with Label 1
c with Label 0

Assembly Code:
MOV a,R[0]
MOV b,R[1]

```
MUL c,R[1]
SUB R[1],R[0]
MOV a,R[1]
MOV d,R[2]
SUB c,R[2]
DIV R[2],R[1]
ADD R[1],R[0]
```