6.837 Computer Graphics
Hierarchical Modeling

Wojciech Matusik, MIT EECS

Some slides from BarbCutler & Jaakko Lehtinen

1

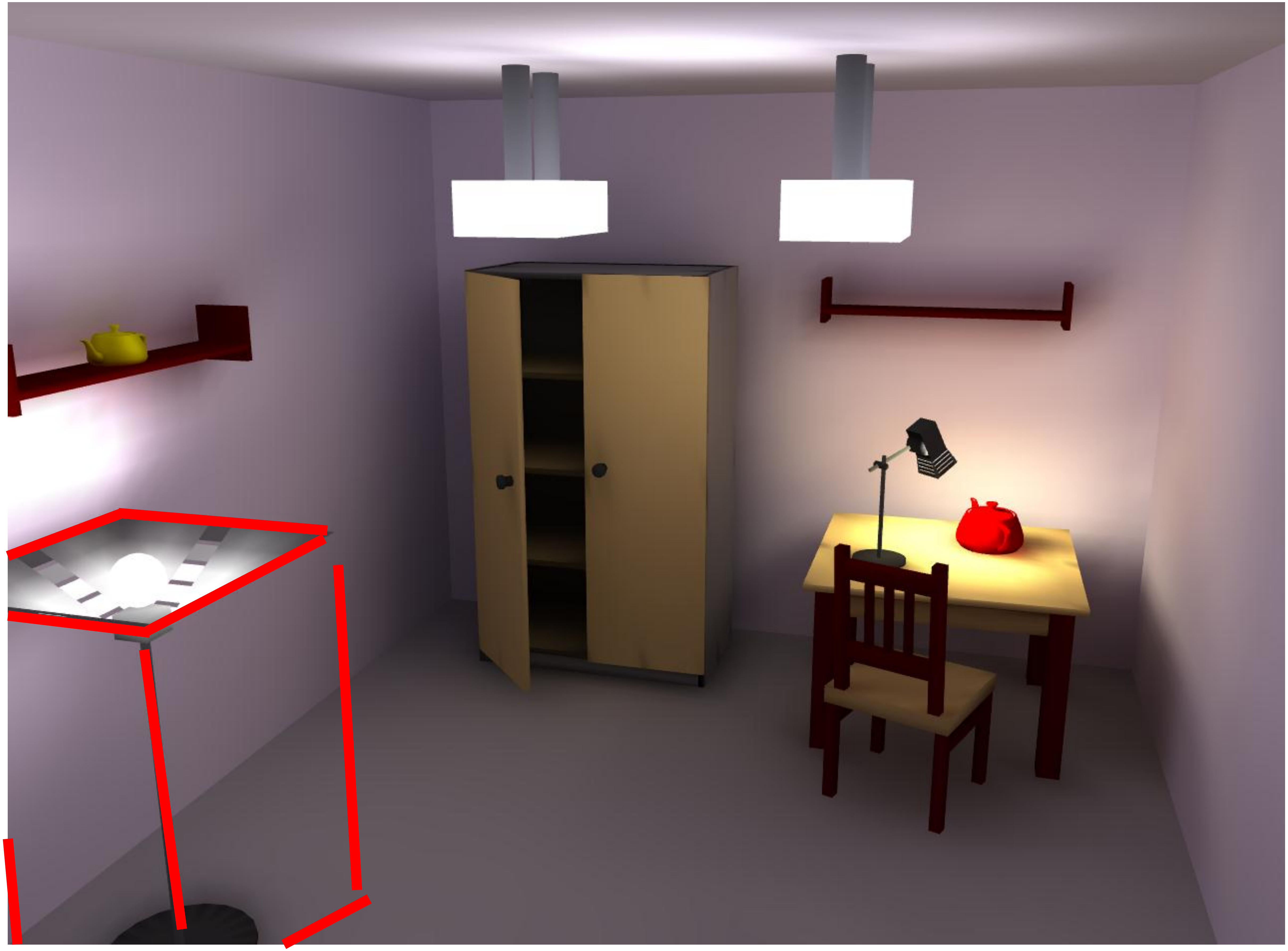
Hierarchical Modeling

- Triangles, parametric curves and surfaces are the building blocks from which more complex real-world objects are modeled.

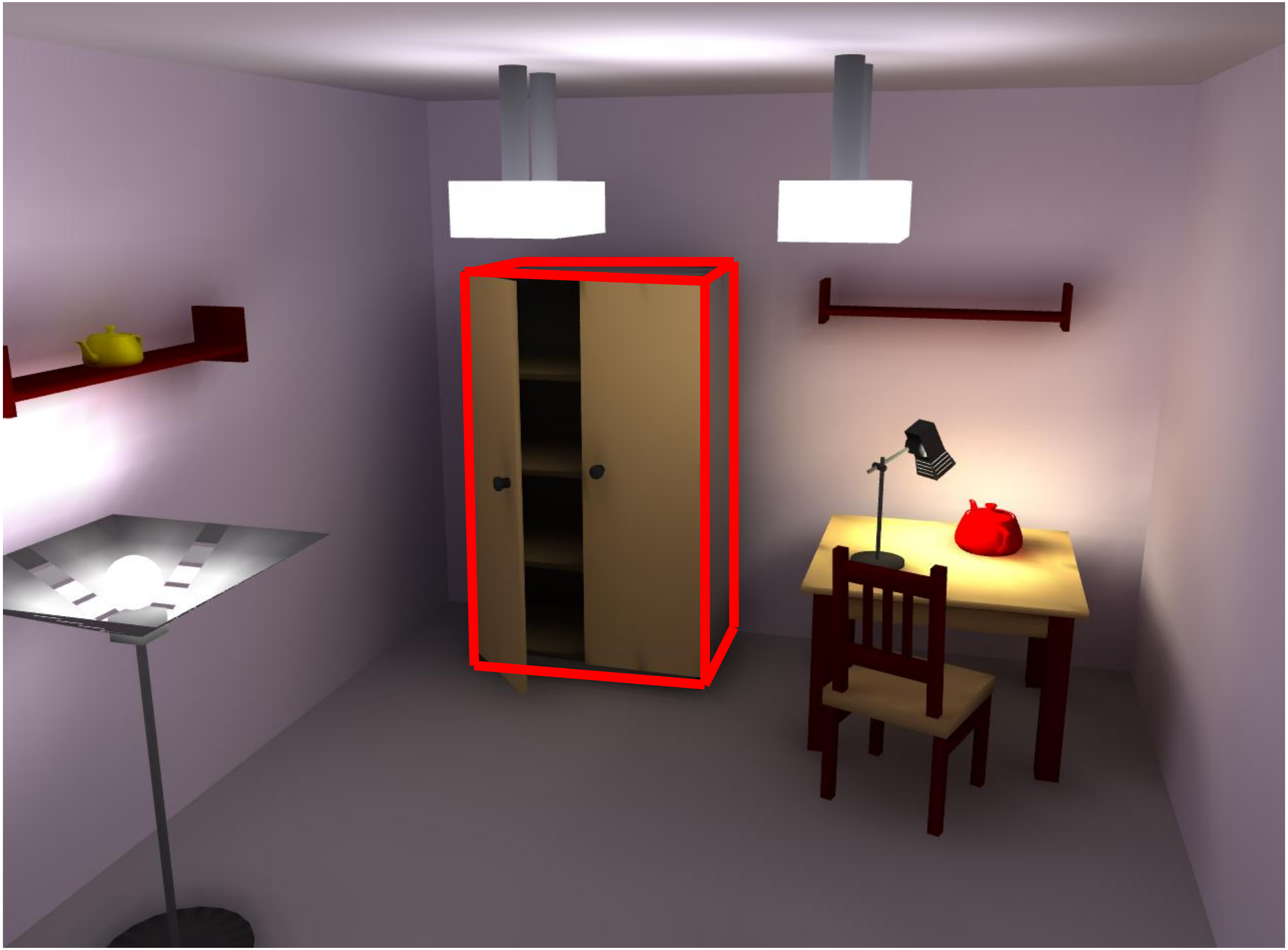
- Hierarchical modeling creates complex real-world objects by combining simple primitive shapes into more complex aggregate objects.

Hierarchical models

Hierarchical models



Hierarchical models



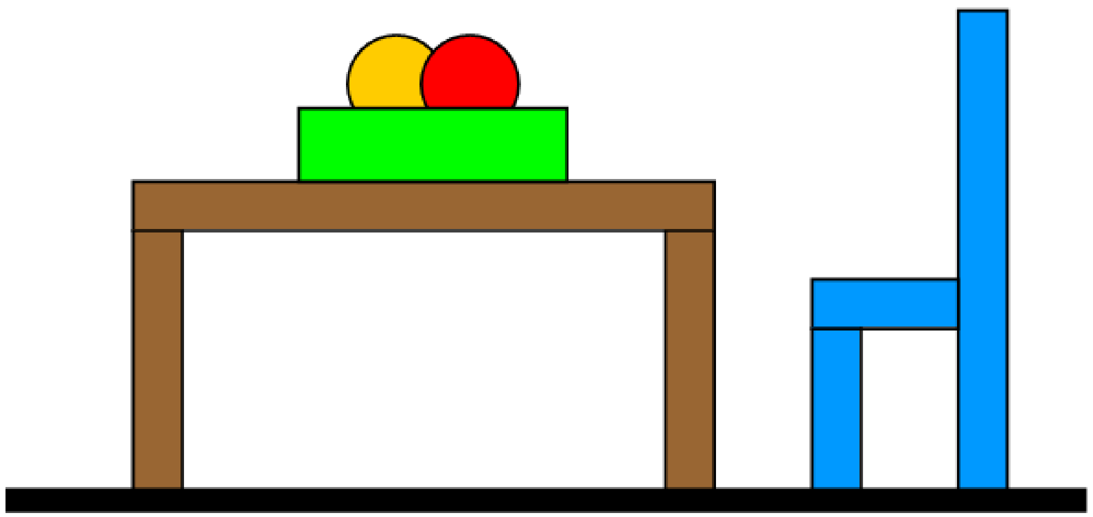
Hierarchical models

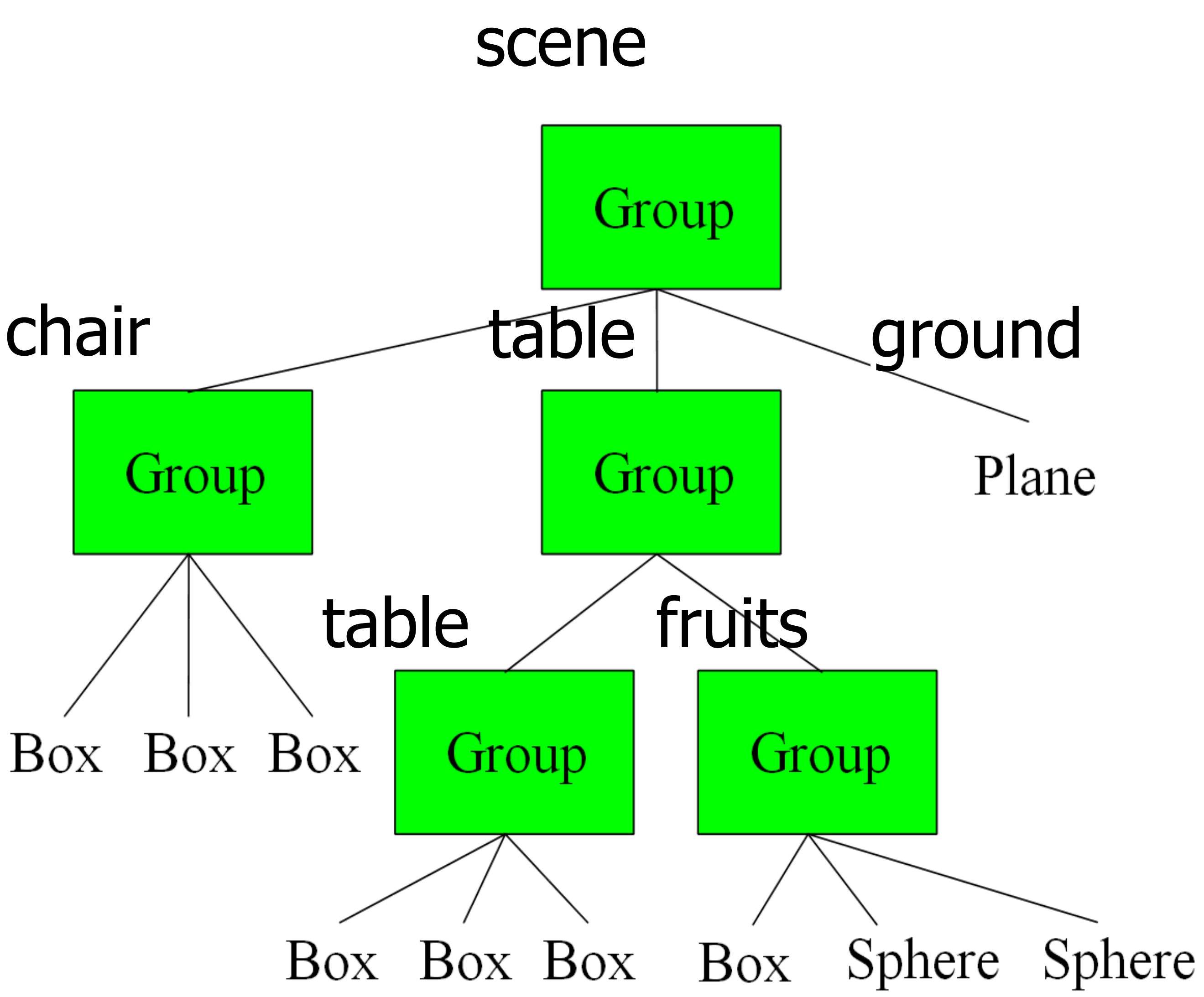
Hierarchical models

Hierarchical models

Hierarchical Grouping of Objects

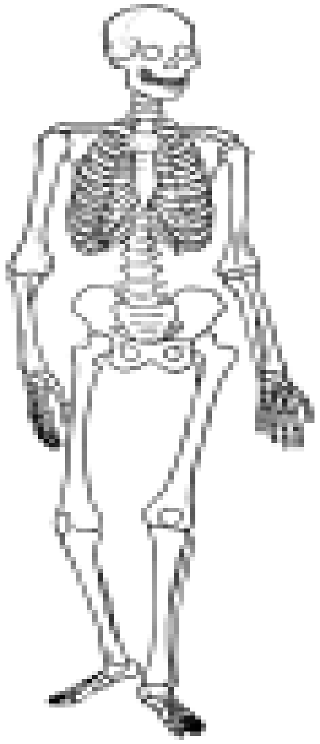
- The "scene graph" represents the logical organization of scene





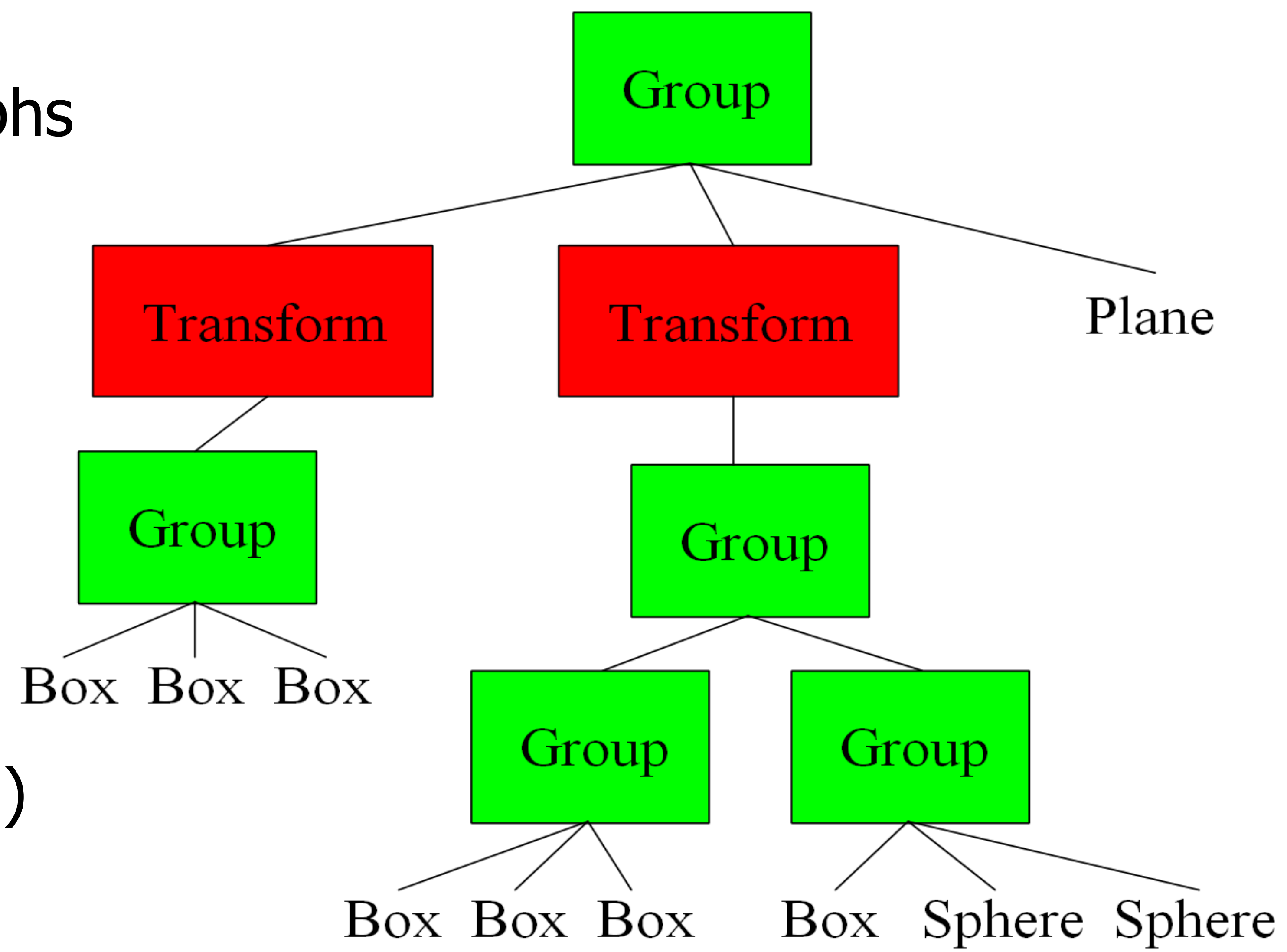
Scene Graph

- Convenient Data structure for scene representation
  - Geometry (meshes, etc.)
  - Transformations
  - Materials, color
  - Multiple instances
- Basic idea: Hierarchical Tree
- Useful for manipulation/animation
  - Also for articulated figures
- Useful for rendering, too
  - Ray tracing acceleration, occlusion culling
  - But note that two things that are close to each other in the tree are NOT necessarily spatially near each other



Scene Graph Representation

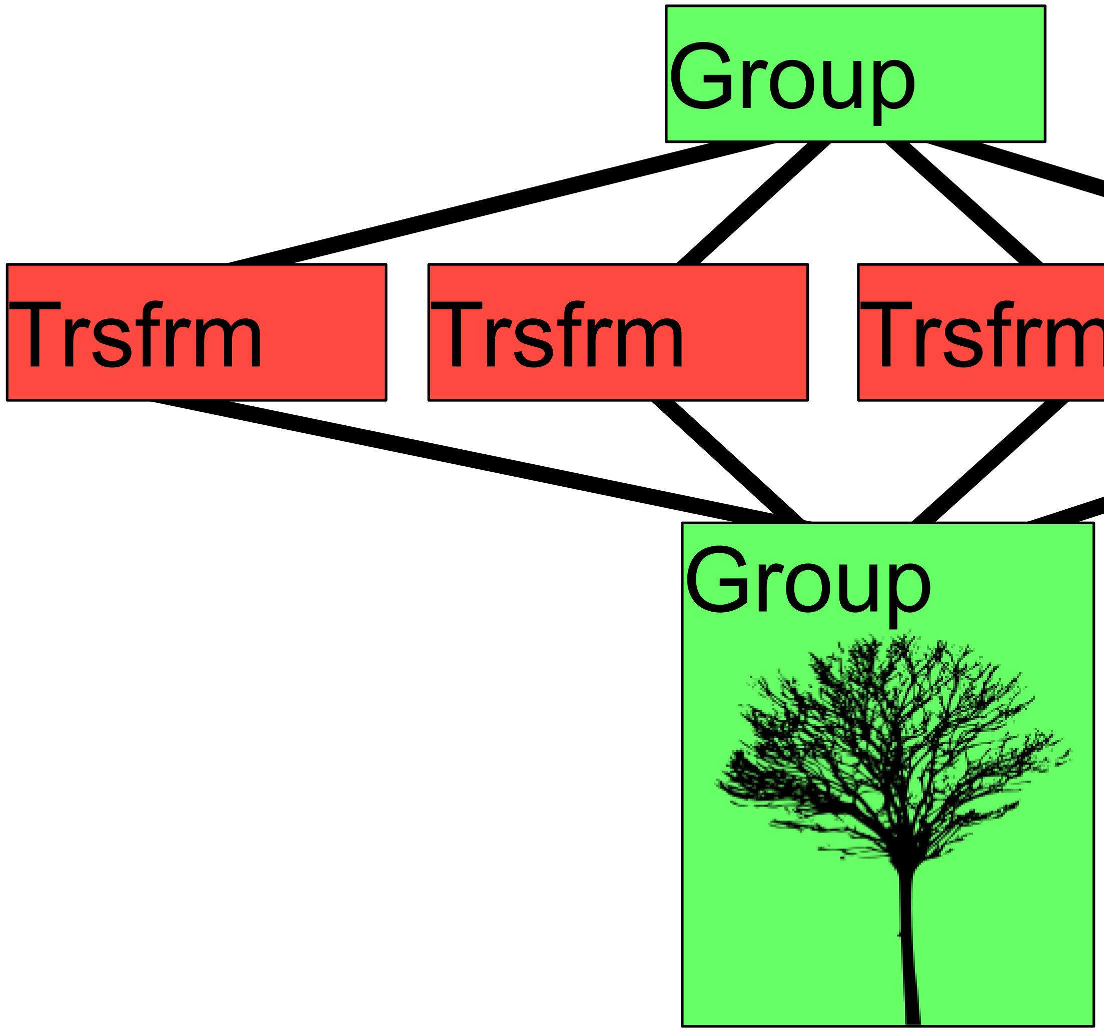
- Basic idea: Tree
- Comprised of several node types
  - Shape: 3D geometric objects
  - Transform: Affect current transformation
  - Property: Color, texture
  - Group: Collection of subgraphs

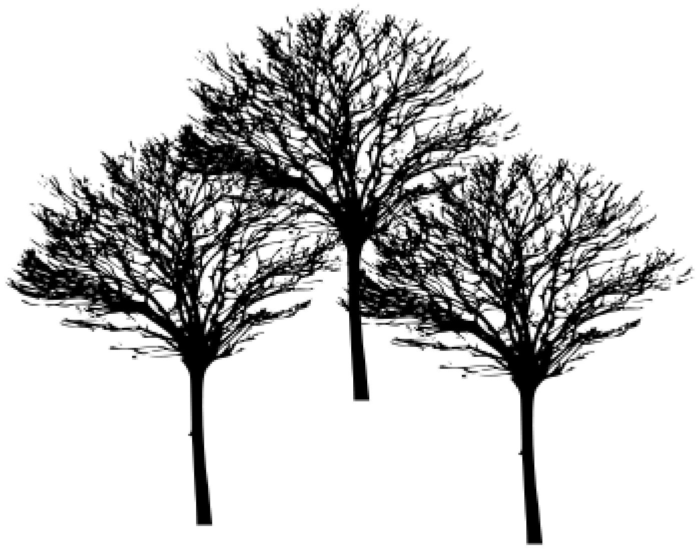


- C++ implementation
  - base class Object
    - children, parent
  - derived classes for each node type (group, transform)

Scene Graph Representation

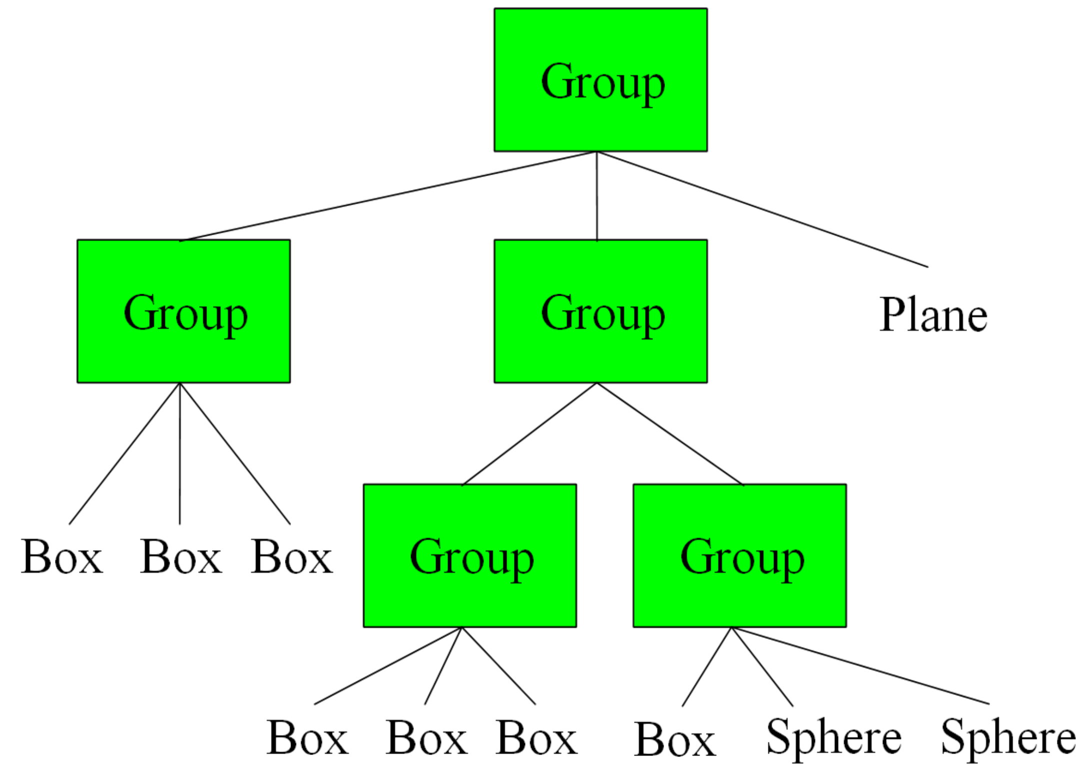
- In fact, generalization of a tree: Directed Acyclic Graph (DAG)
  - Means a node can have multiple parents, but cycles are not allowed
- Why? Allows multiple instantiations
  - Reuse complex hierarchies many times in the scene using different transformations (example: a tree)
    - Of course, if you only want to reuse meshes, just load the mesh once and make several geometry nodes point to the same data

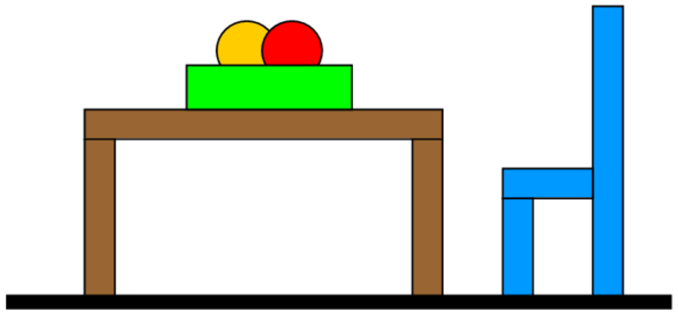




Simple Example with Groups

```
Group {
    numObjects 3
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Box { <BOX PARAMS> }
            Sphere { <SPHERE PARAMS> }
            Sphere { <SPHERE PARAMS> } } }
    Plane { <PLANE PARAMS> } }
```

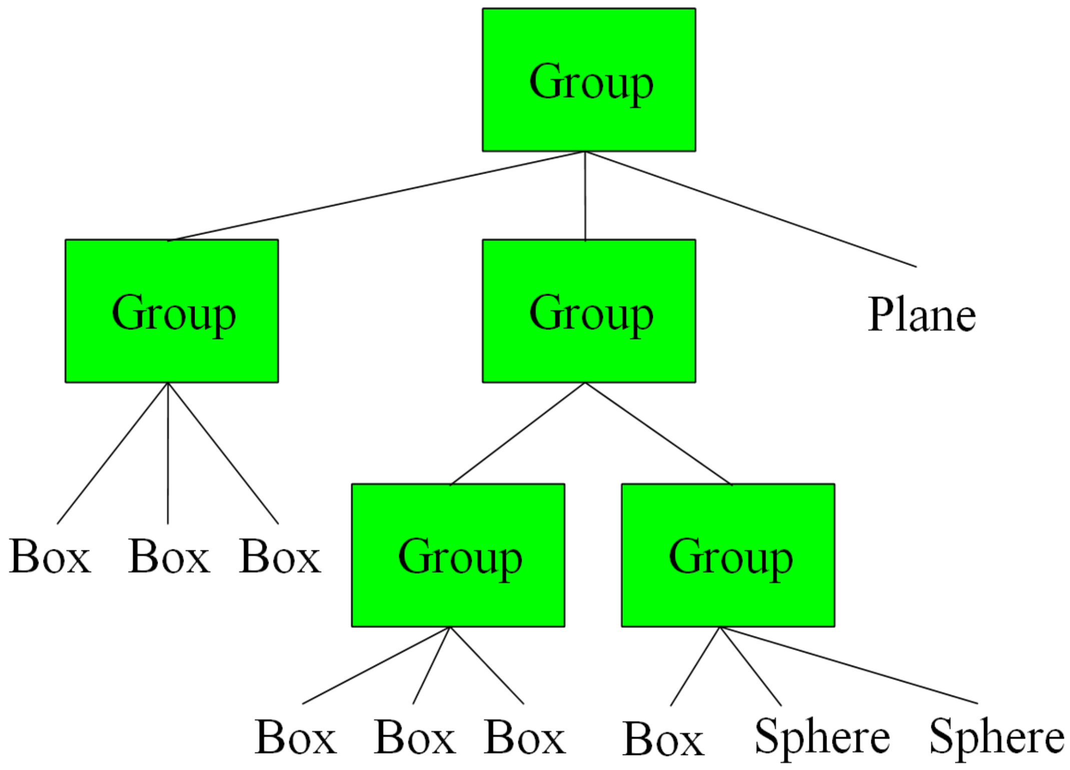


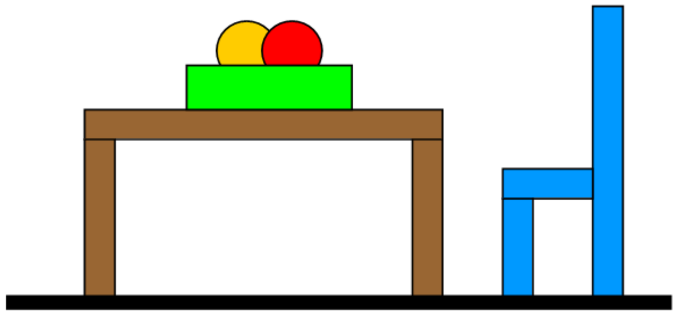


Text format is fictitious, better to use XML in real applications

Simple Example with Groups

```
Group {
     numObjects 3
     Group {
          numObjects 3
          Box { <BOX PARAMS> }
          Box { <BOX PARAMS> }
          Box { <BOX PARAMS> } }
     Group {
          numObjects 2
          Group {
               Box { <BOX PARAMS> }
               Box { <BOX PARAMS> }
               Box { <BOX PARAMS> } }
          Group {
               Box { <BOX PARAMS> }
               Sphere { <SPHERE PARAMS> }
               Sphere { <SPHERE PARAMS> } } }
     Plane { <PLANE PARAMS> } }
```

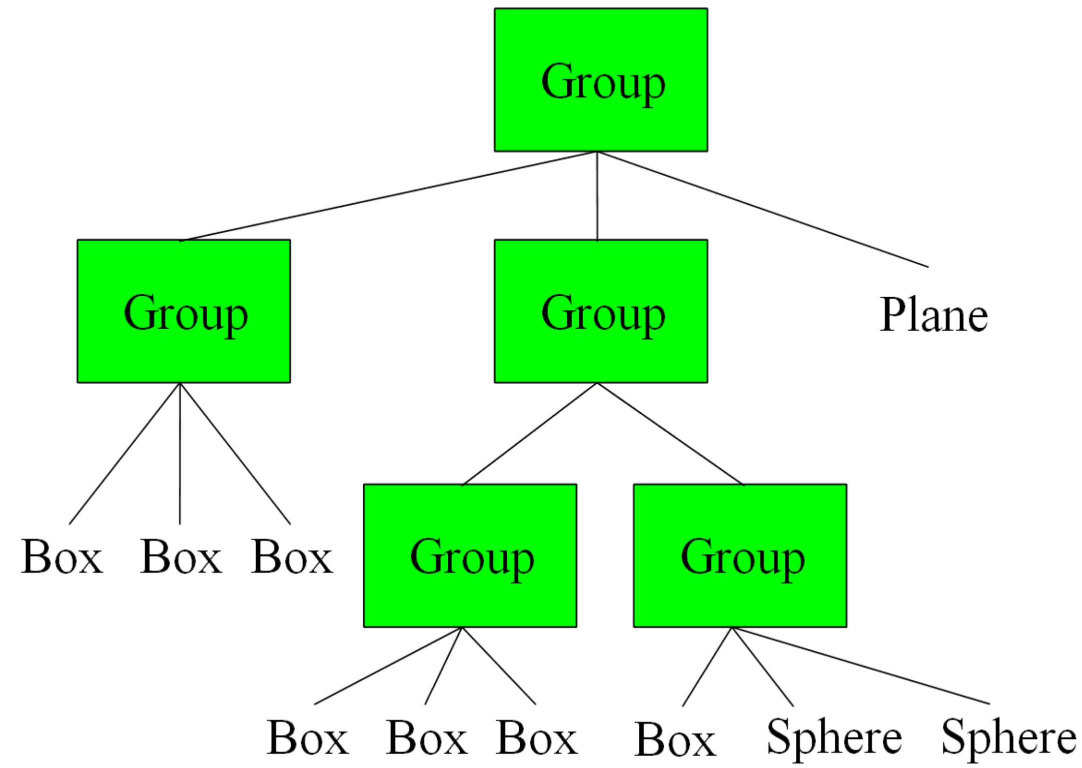


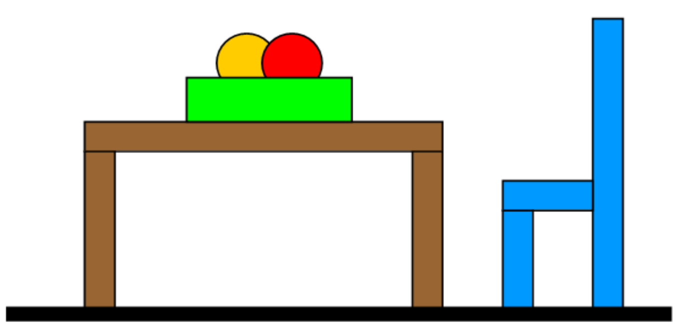


Text format is fictitious, better to use XML in real applications

Simple Example with Groups

```
Group {
    numObjects 3
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Box { <BOX PARAMS> }
            Sphere { <SPHERE PARAMS> }
            Sphere { <SPHERE PARAMS> } } }
    Plane { <PLANE PARAMS> } }
```



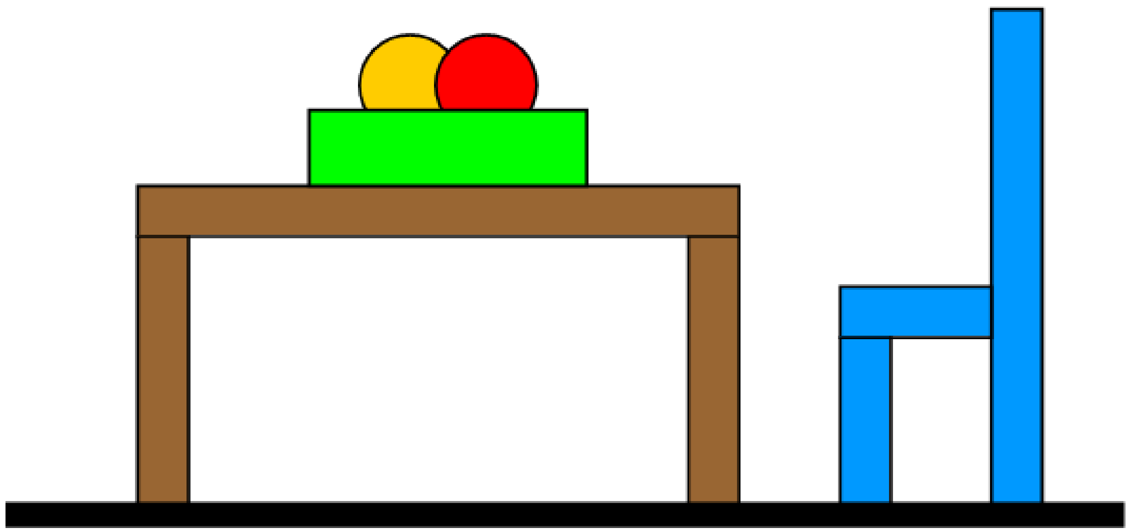


Here we have only simple shapes, but easy to add a “Mesh” node whose parameters specify an .OBJ to load (say)

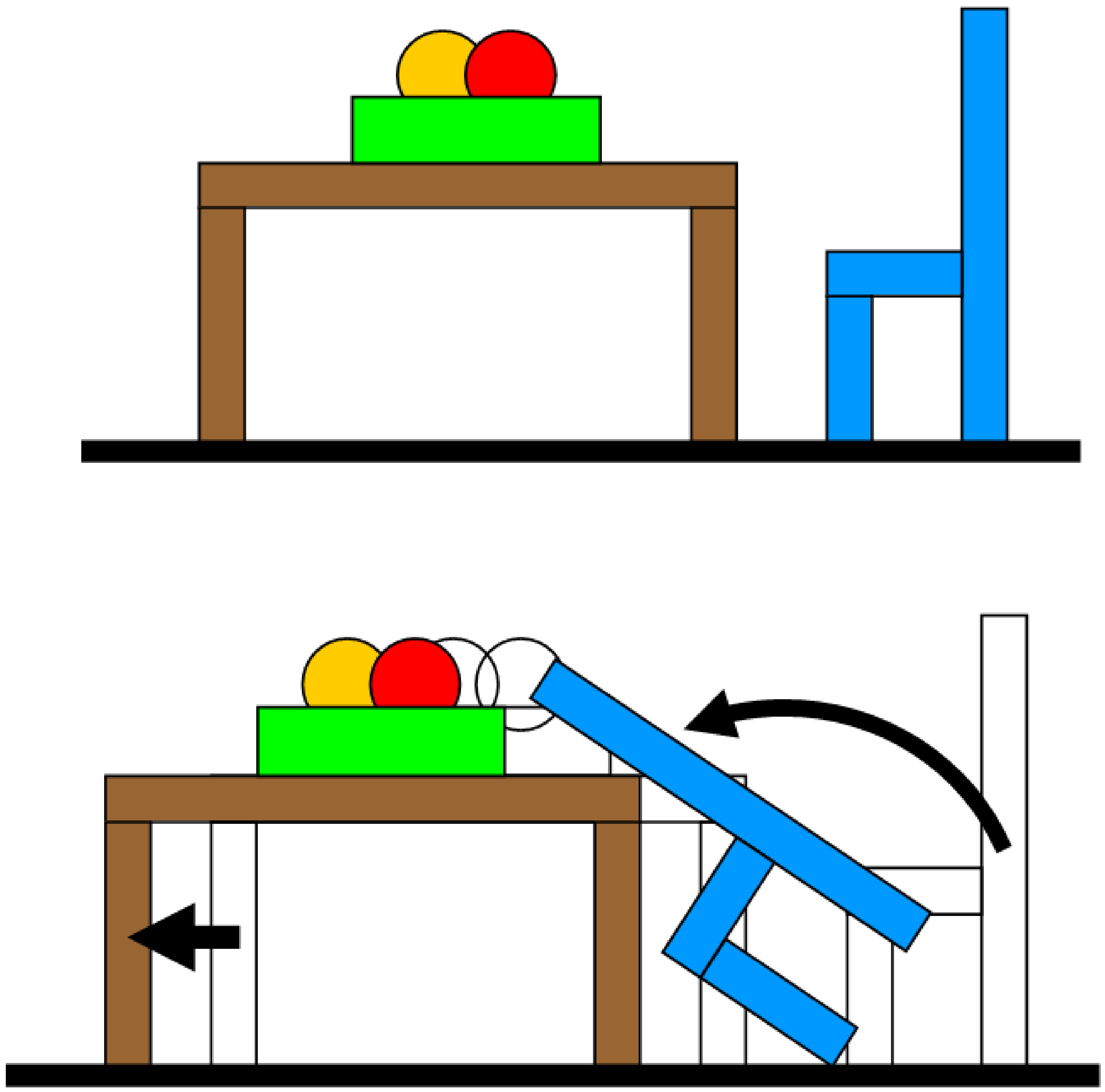
Adding Attributes (Material, etc.)

```
Group {
    numObjects 3
    Material { <BLUE> }
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Material { <BROWN> }
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Material { <GREEN> }
            Box { <BOX PARAMS> }
            Material { <RED> }
            Sphere { <SPHERE PARAMS> }
            Material { <ORANGE> }
            Sphere { <SPHERE PARAMS> } } }

    Plane { <PLANE PARAMS> } }
```



Adding Transformations

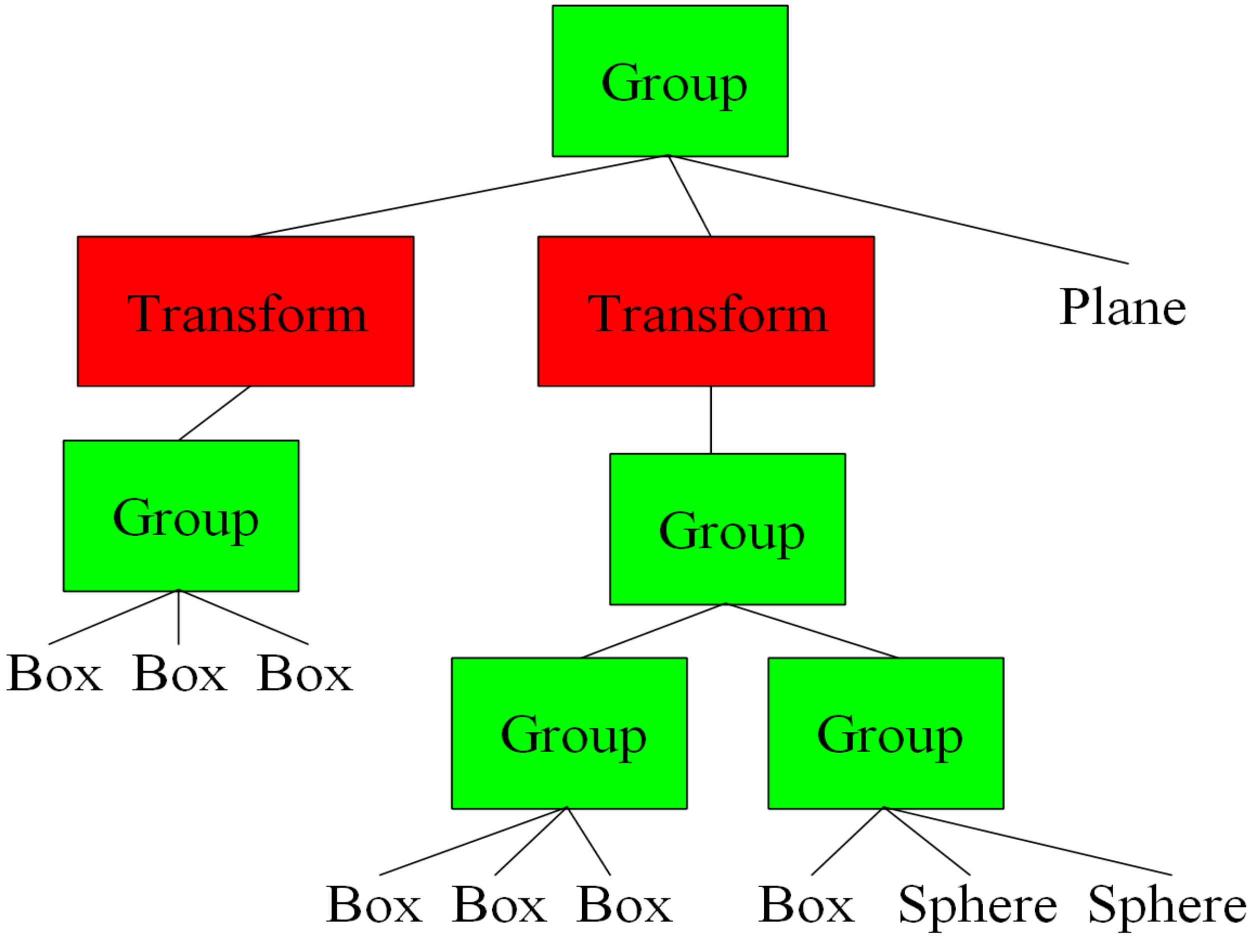


Questions?

Scene Graph Traversal

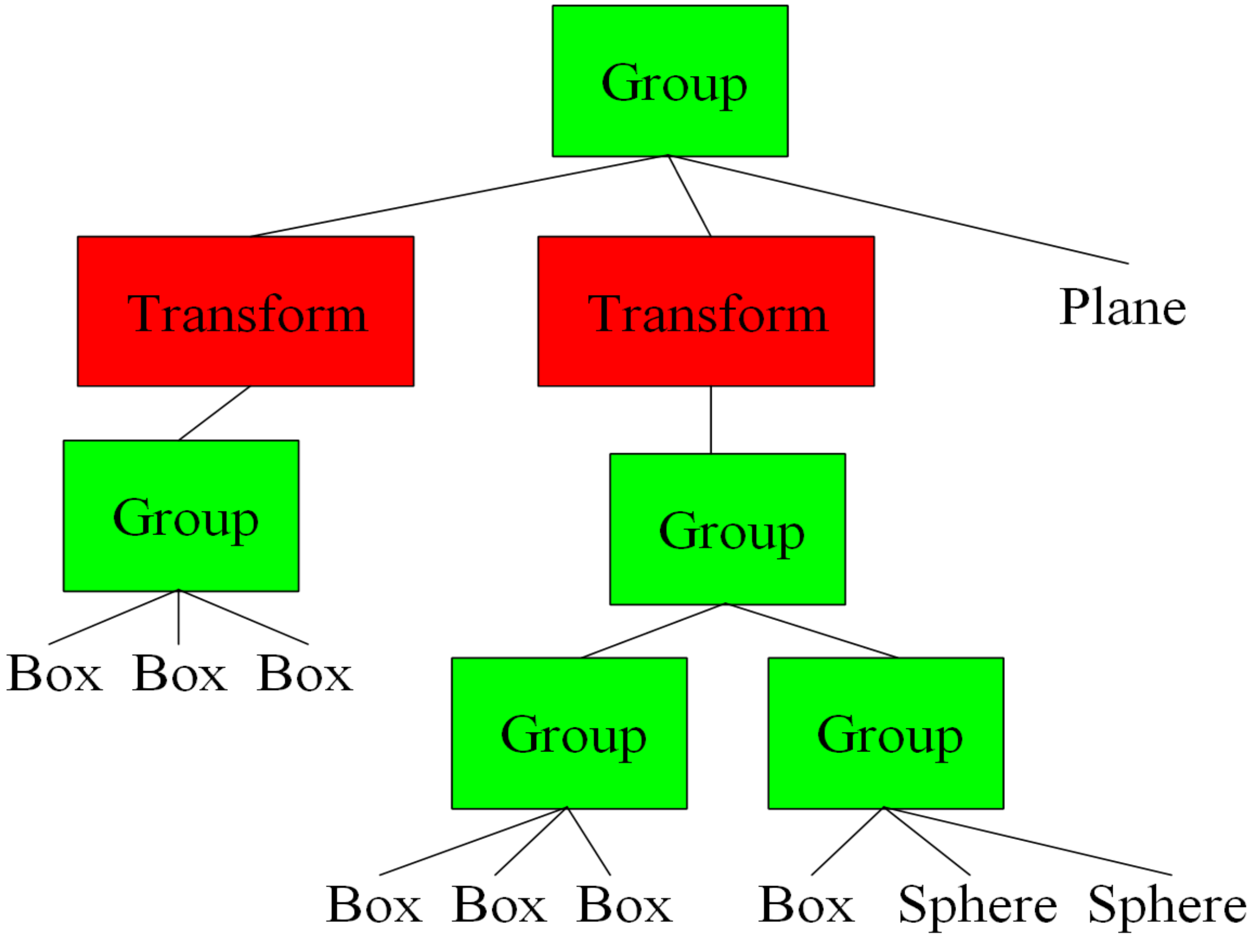
- Depth first recursion
  - Visit node, then visit subtrees (top to bottom, left to right)
  - When visiting a geometry node: Draw it!

- How to handle transformations?
  - Remember, transformations are always specified in coordinate system of the parent



Scene Graph Traversal

- How to handle transformations?
  - Traversal algorithm keeps a **transformation state S** (a 4x4 matrix)
    - from world coordinates
    - Initialized to identity in the beginning
  - Geometry nodes always drawn using current **S**
  - When visiting a transformation node **T**: multiply current state **S** with **T**, then visit child nodes
    - Has the effect that nodes below will have new transformation
  - When all children have been visited, **undo the effect of T!**



Recall frames

- An object frame has coordinates O in the world (of course O is also our 4x4 matrix)

$$\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$$

- Then we are given coordinates c in the object frame

$$\vec{\mathbf{o}}^t \mathbf{c} = \vec{\mathbf{w}}^t O \mathbf{c}$$

- Indeed we need to apply matrix O to all objects

Frames and hierarchy

- Matrix *M1* to go from world to torso $\vec{\mathbf{t}}^t = \vec{\mathbf{w}}^t M_1$
- Matrix *M2* to go from torso to arm $\vec{\mathbf{a}}^t = \vec{\mathbf{t}}^t M_2$

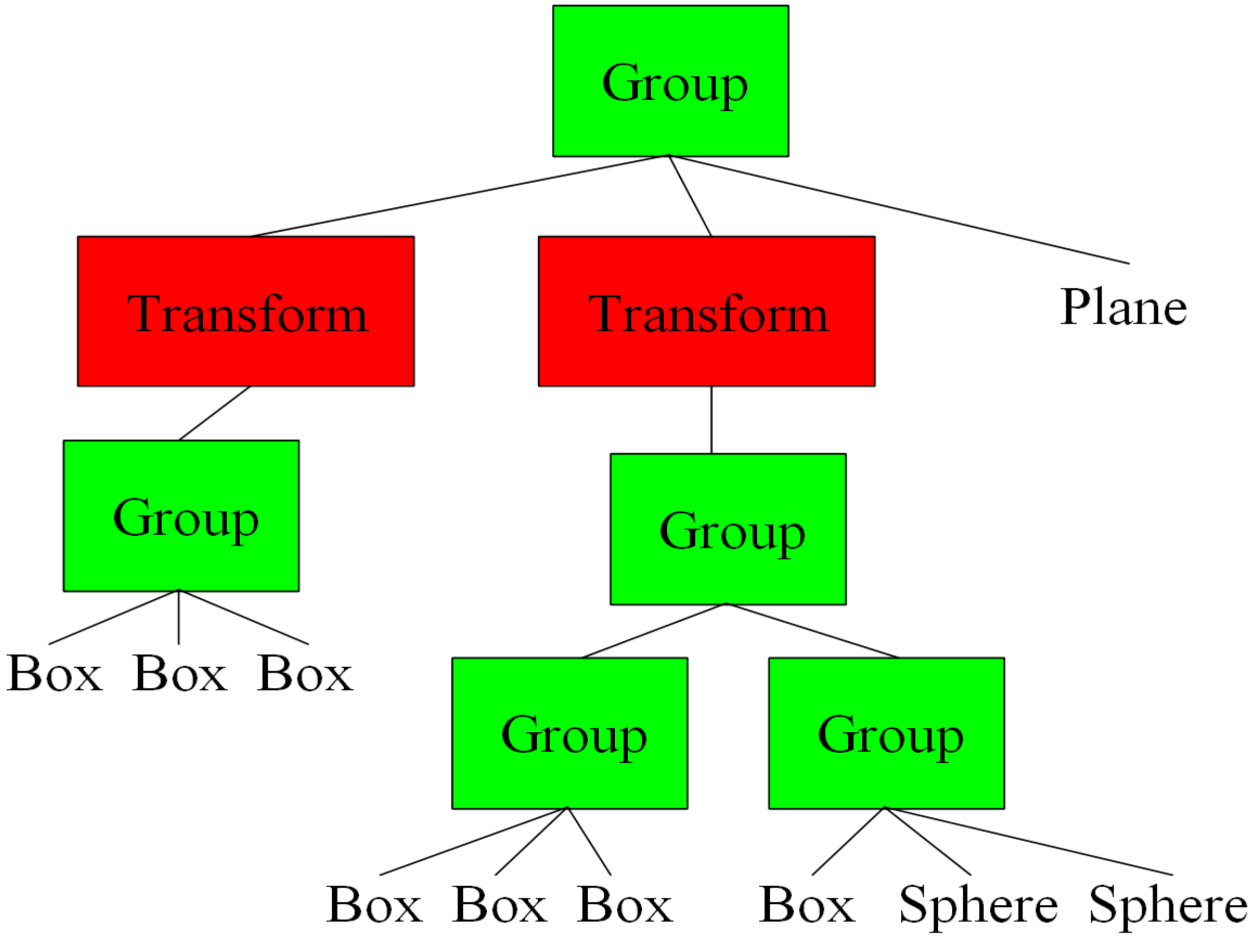
- How do you go from arm coordinates to world?

$$\vec{\mathbf{a}}^t \mathbf{c} = \vec{\mathbf{t}}^t M_2 \mathbf{c} = \vec{\mathbf{w}}^t M_1 M_2 \mathbf{c}$$

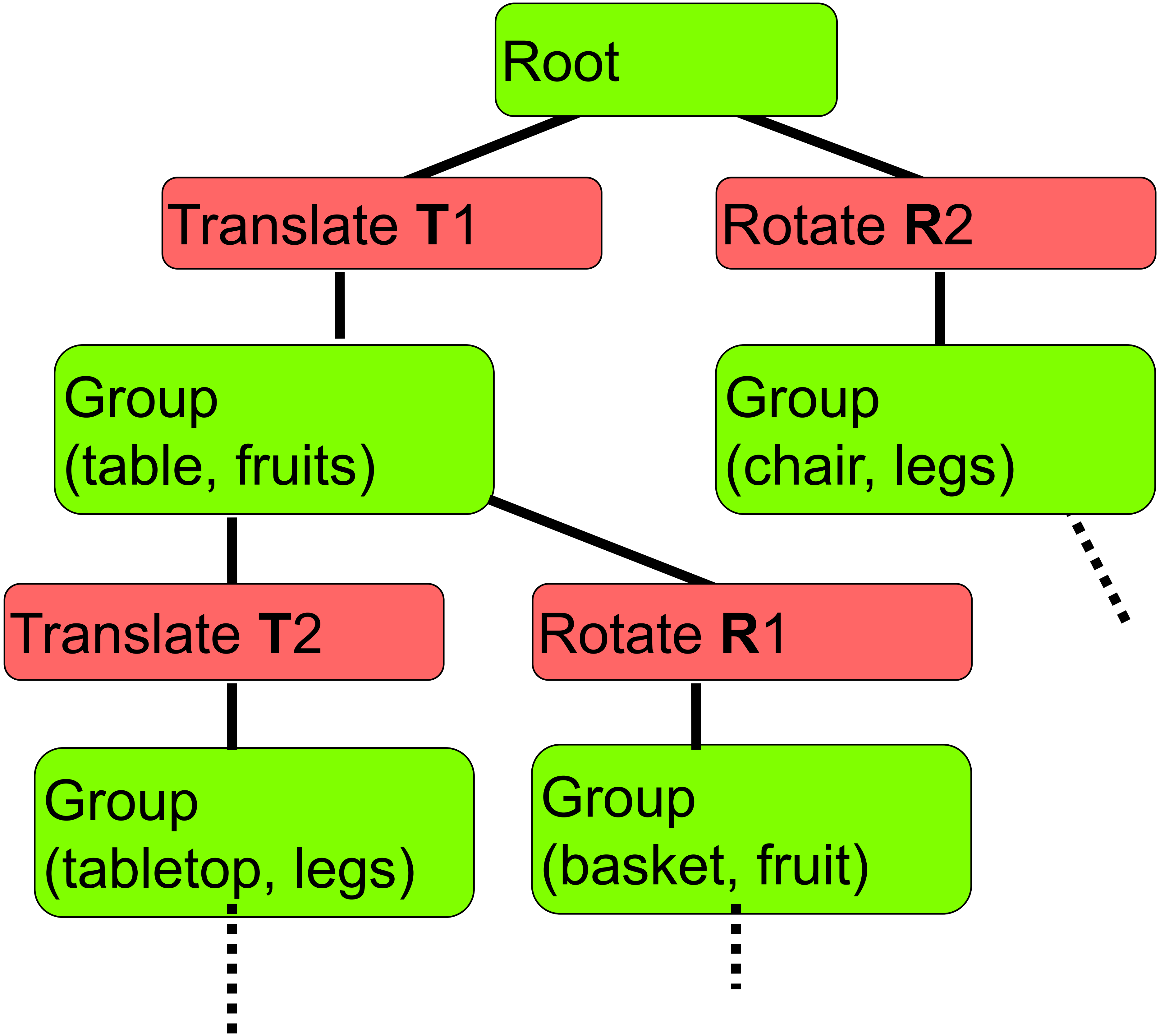
- We can concatenate the matrices
- Matrices for the lower hierarchy nodes go to the right

Recap: Scene Graph Traversal

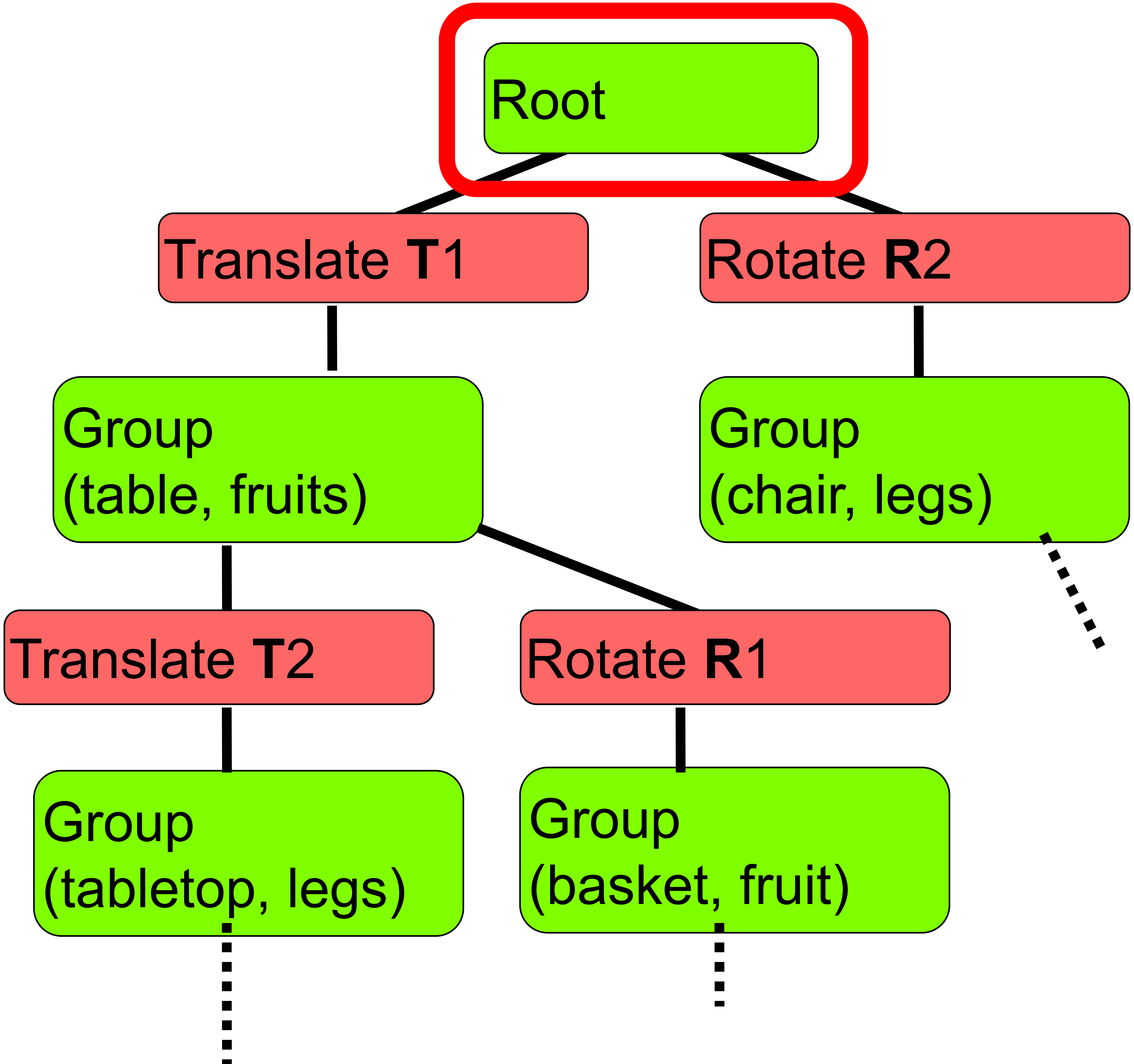
- How to handle transformations?
  - Traversal algorithm keeps a **transformation state S** (a 4x4 matrix)
    - from world coordinates
    - Initialized to identity in the beginning
  - Geometry nodes always drawn using current **S**
  - When visiting a transformation node **T**: multiply current state **S** with **T**, then visit child nodes
    - Has the effect that nodes below will have new transformation
  - When all children have been visited, **undo the effect of T!**



Traversal Example

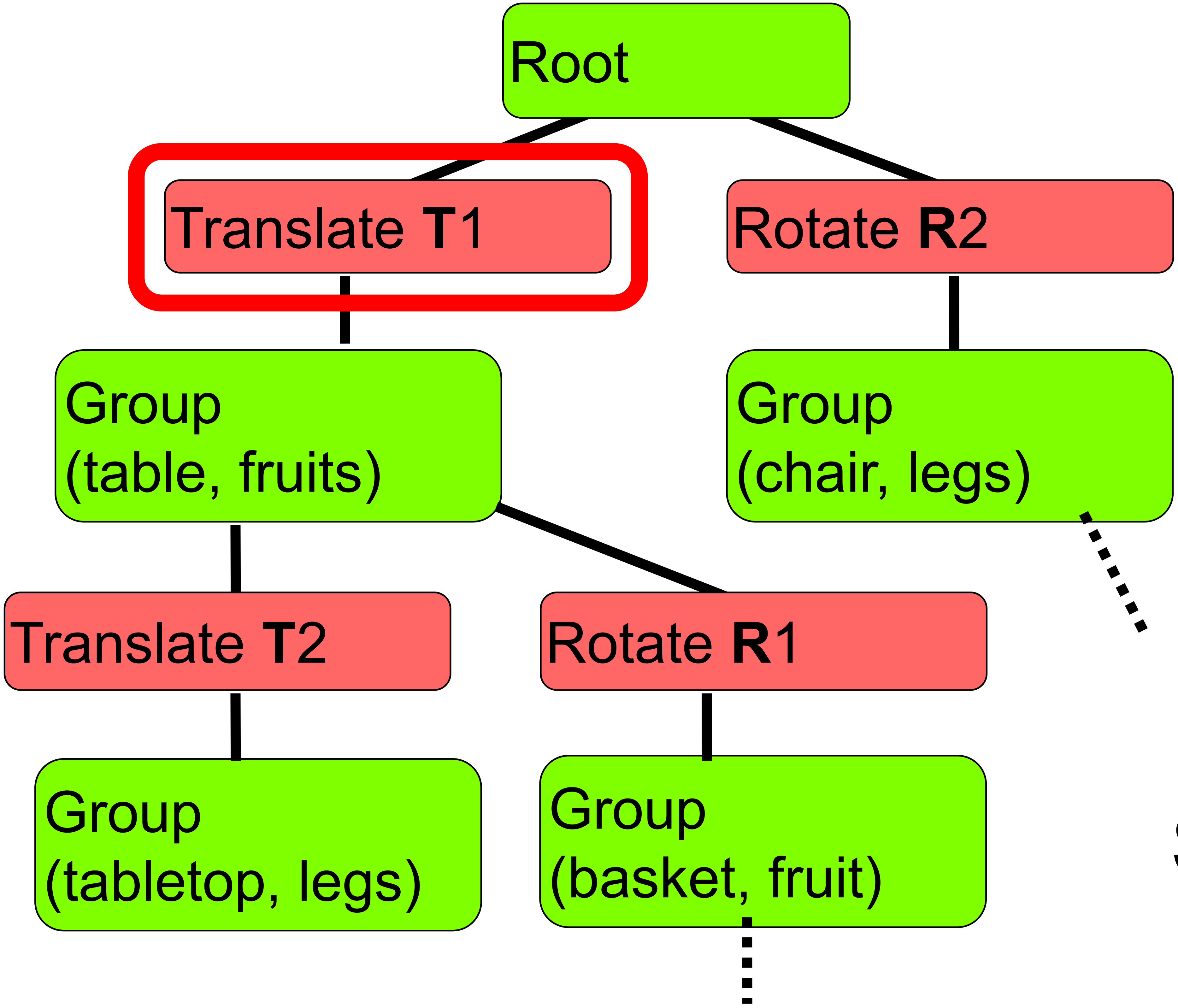


Traversal Example



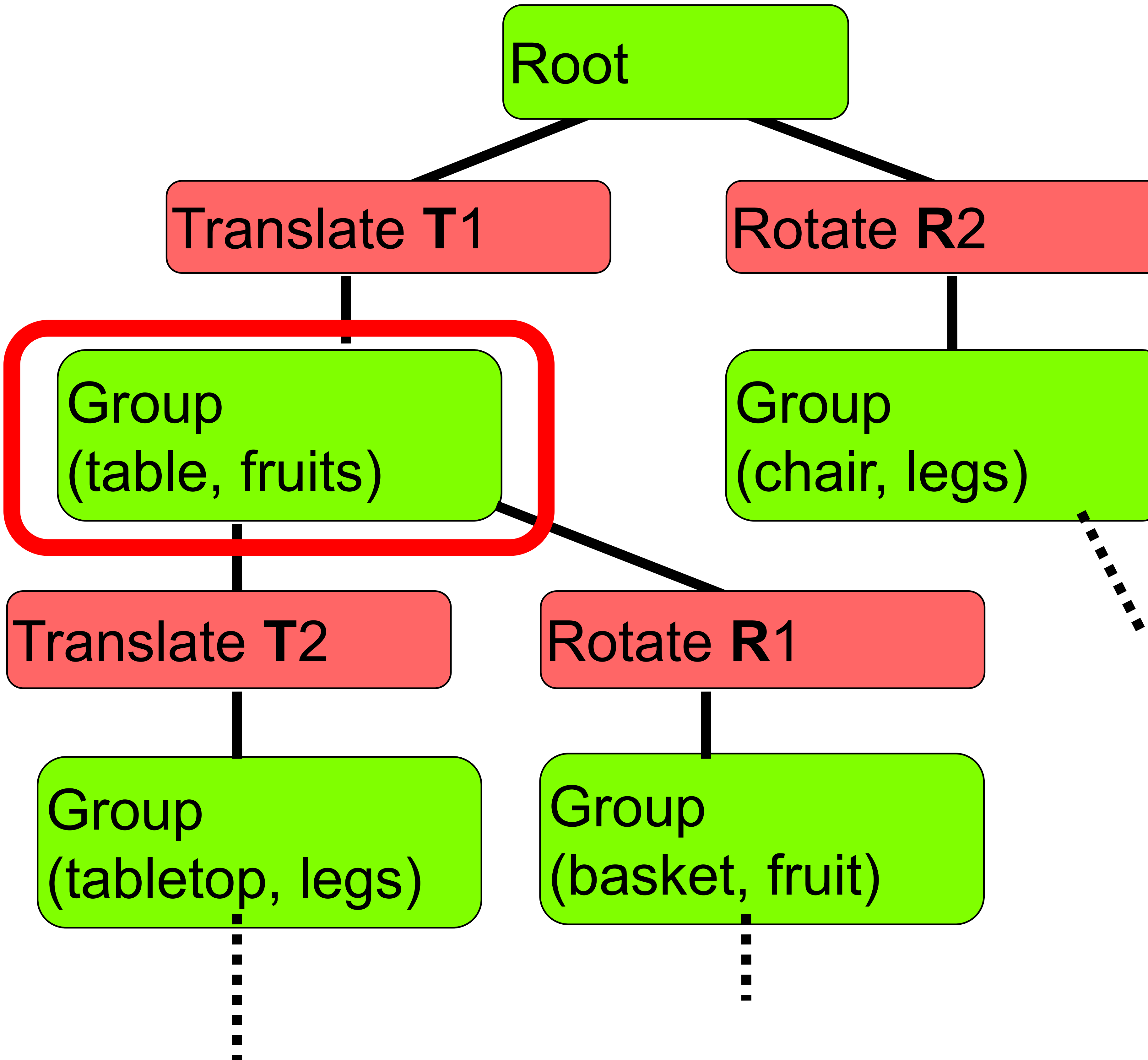
S = **I**

Traversal Example



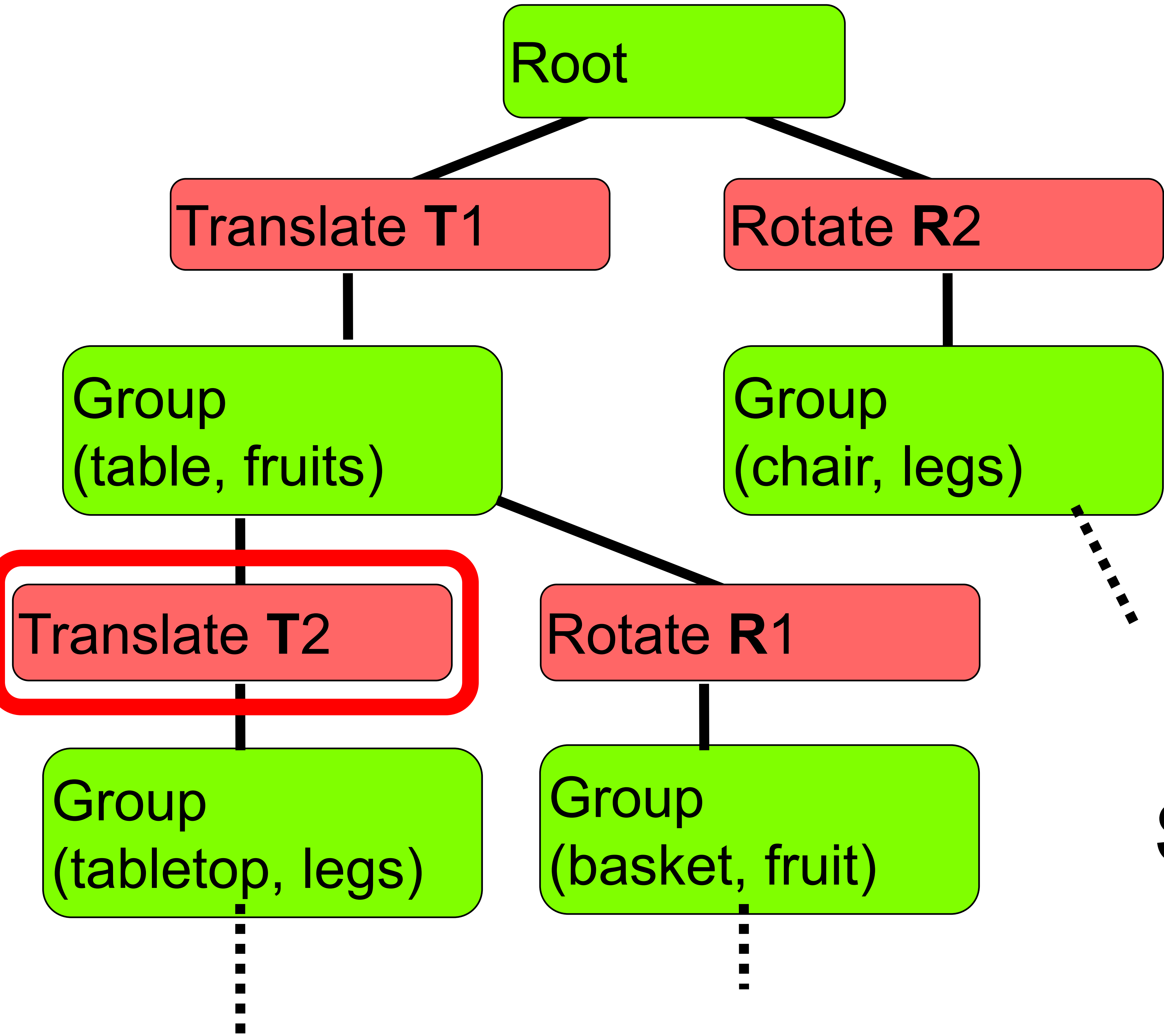
S = **T1**

Traversal Example



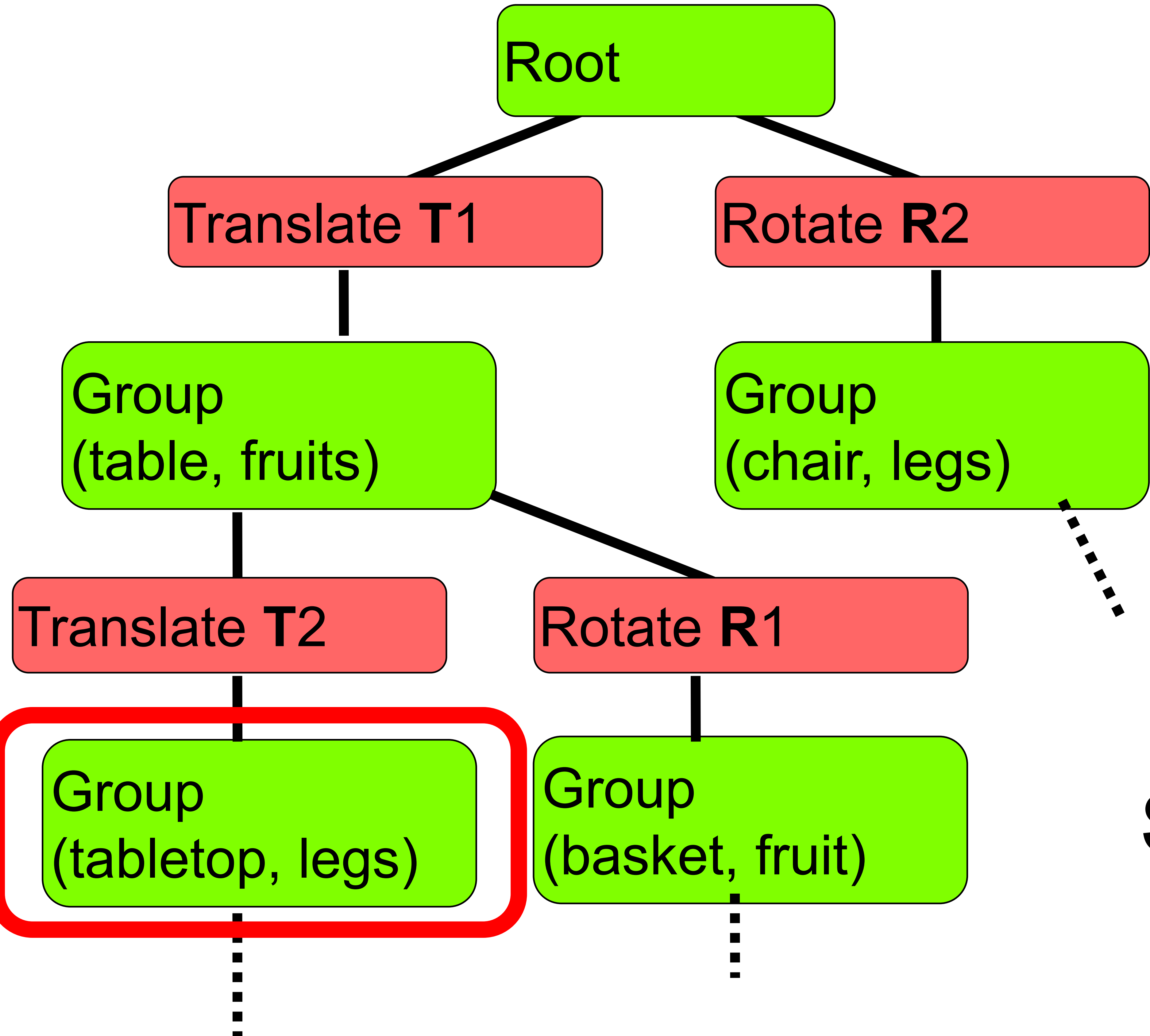
S = **T1**

Traversal Example



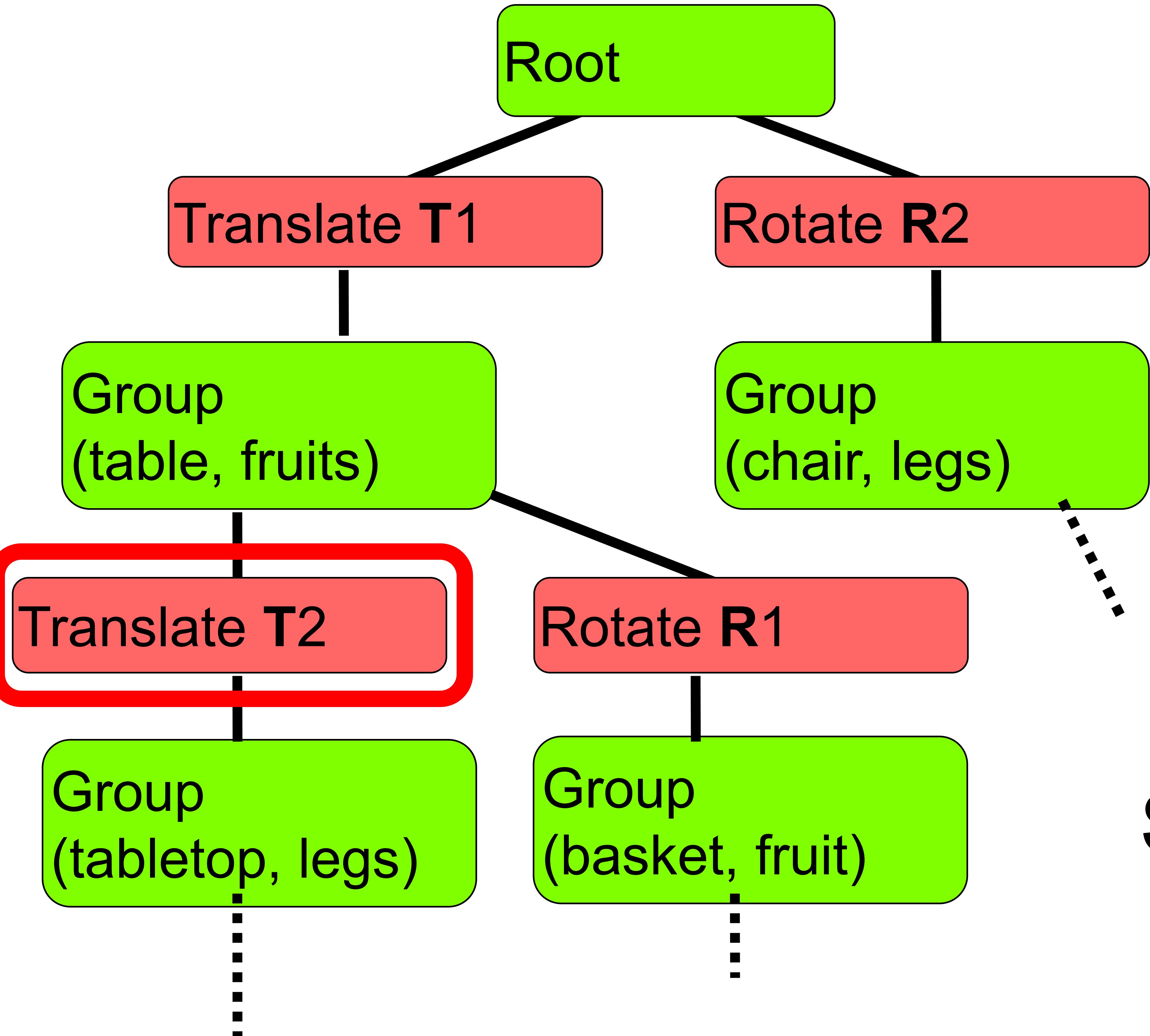
S = T1 T2

Traversal Example



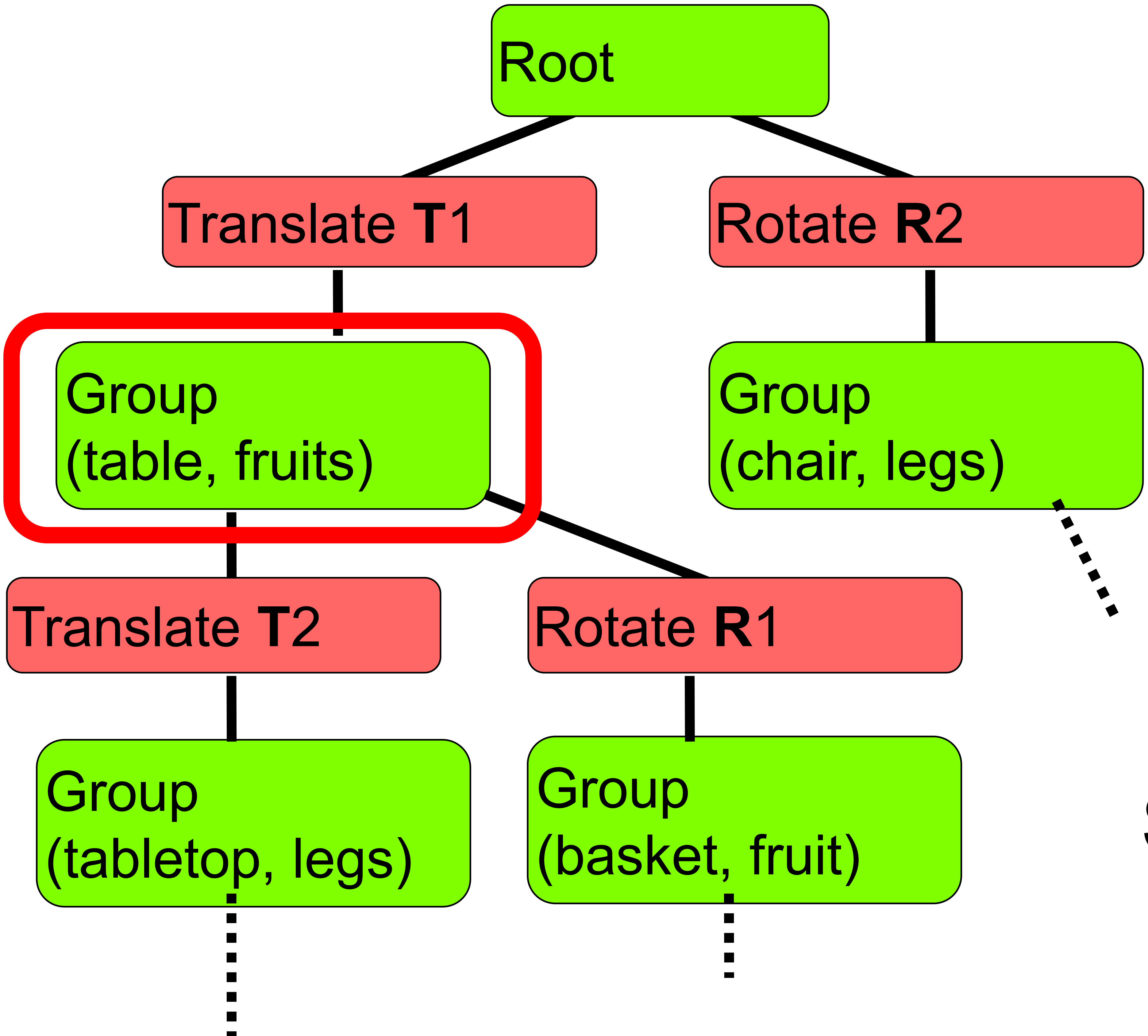
$$\mathbf{S} = \mathbf{T1\ T2}$$

Traversal Example



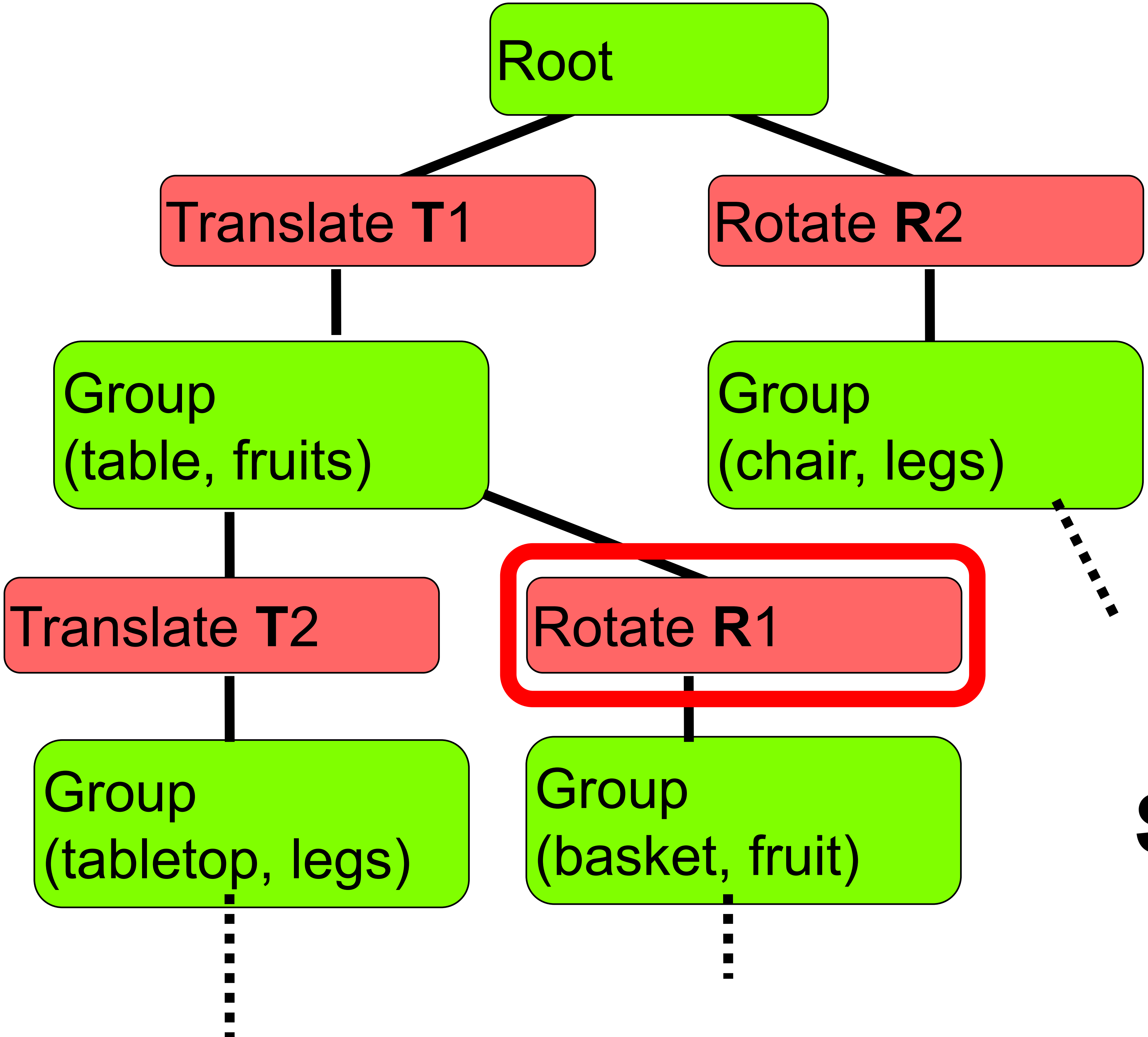
S = **T1 T2**

Traversal Example



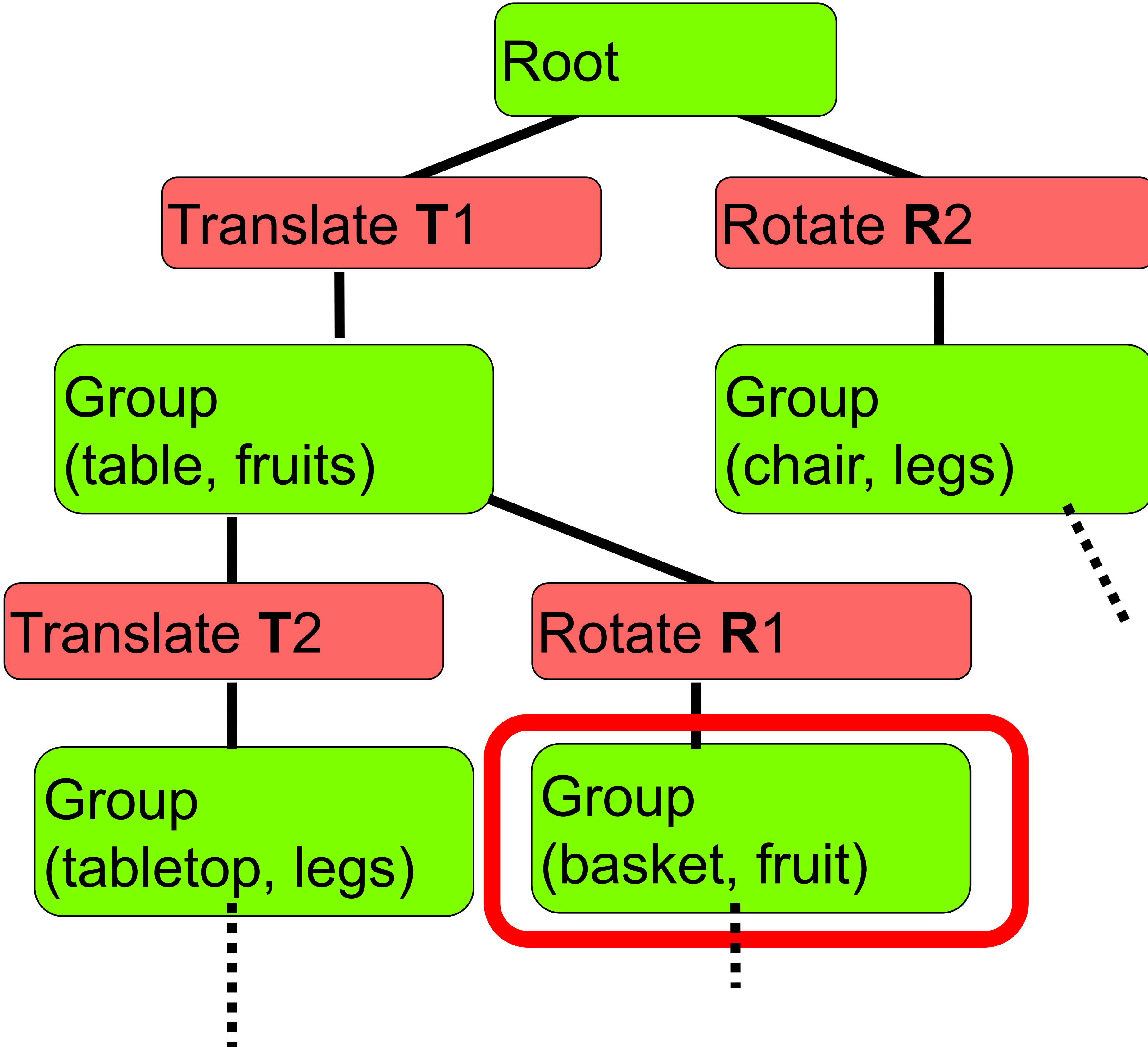
S = **T1**

Traversal Example



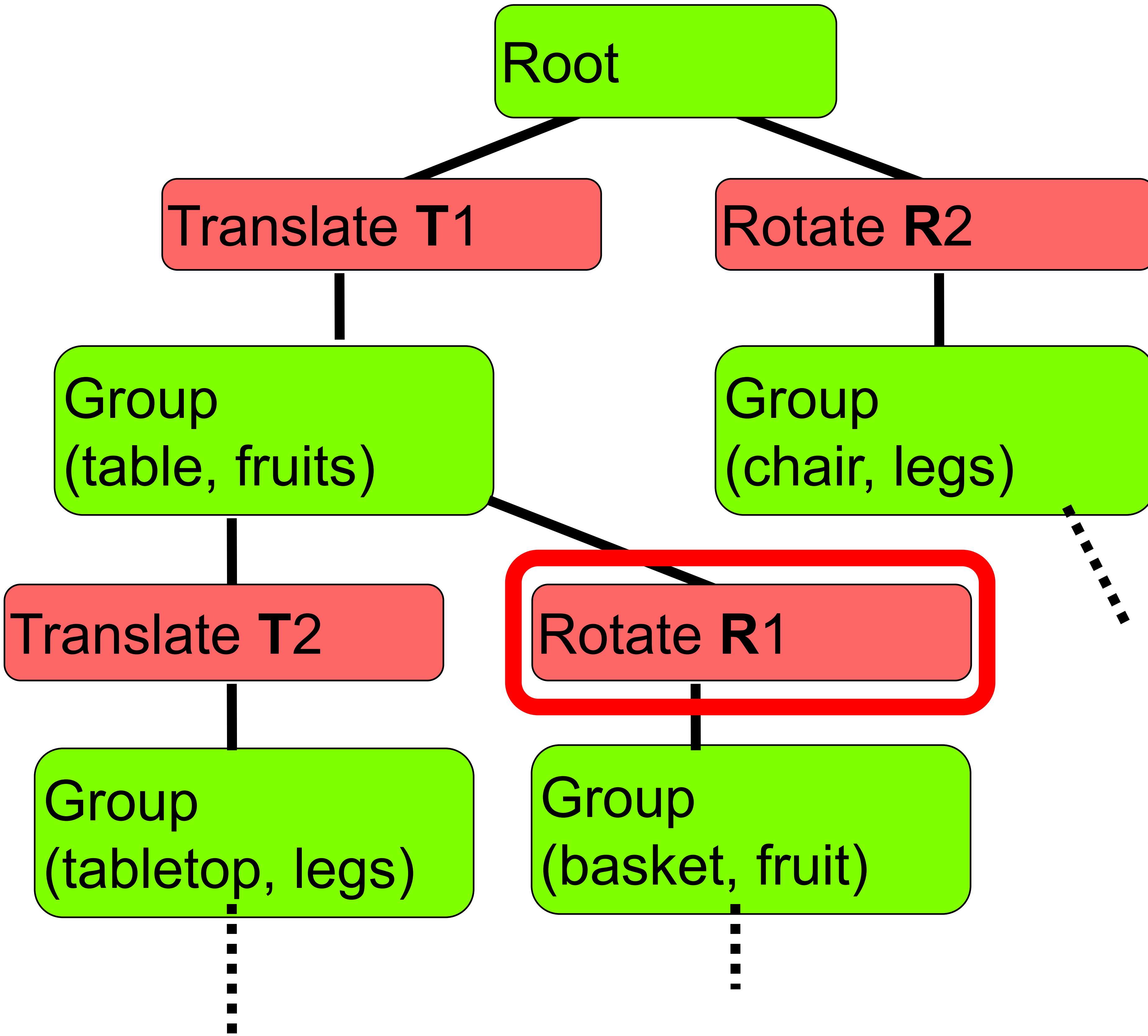
S = T1 R1

Traversal Example



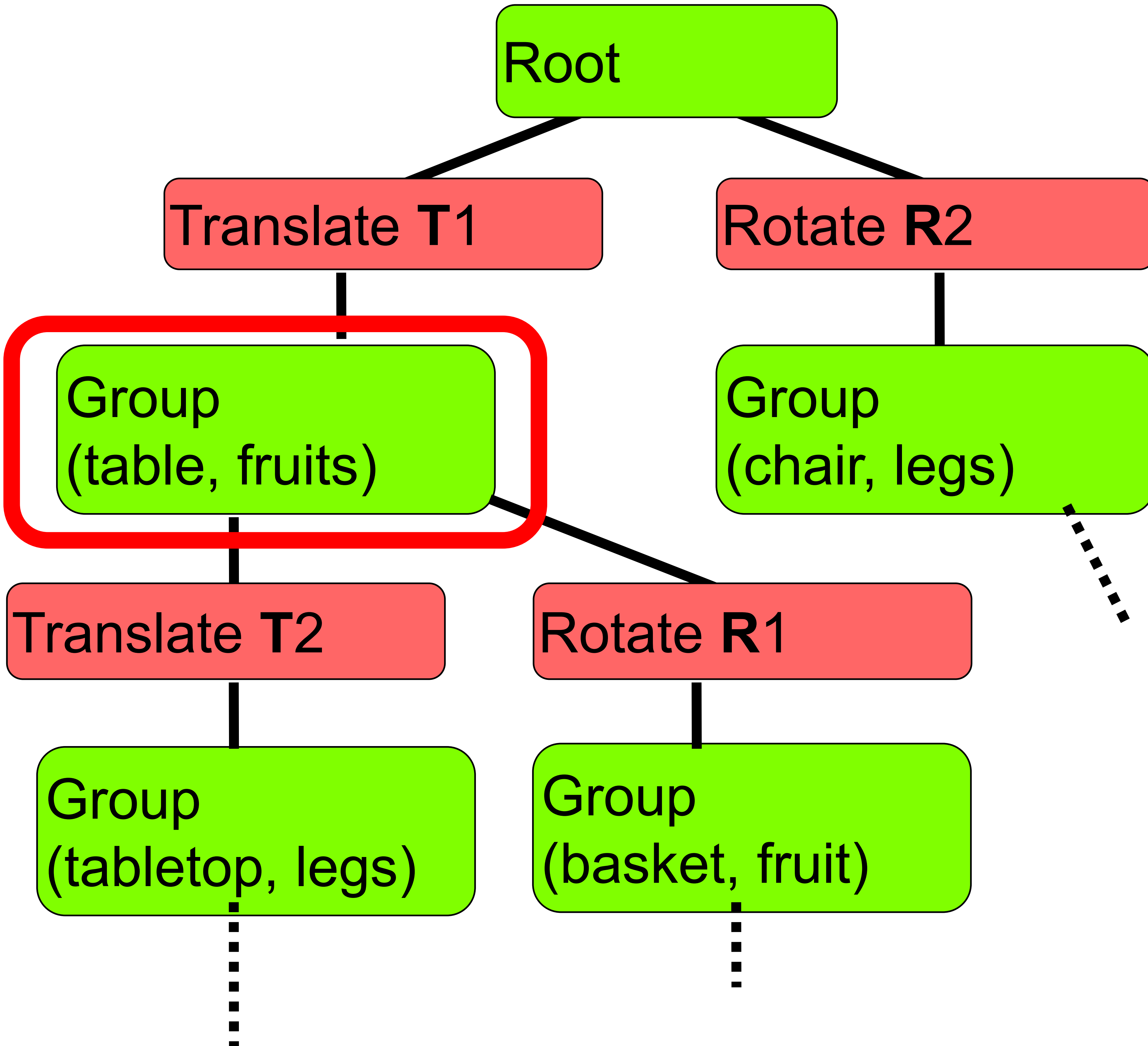
S = T1 R1

Traversal Example



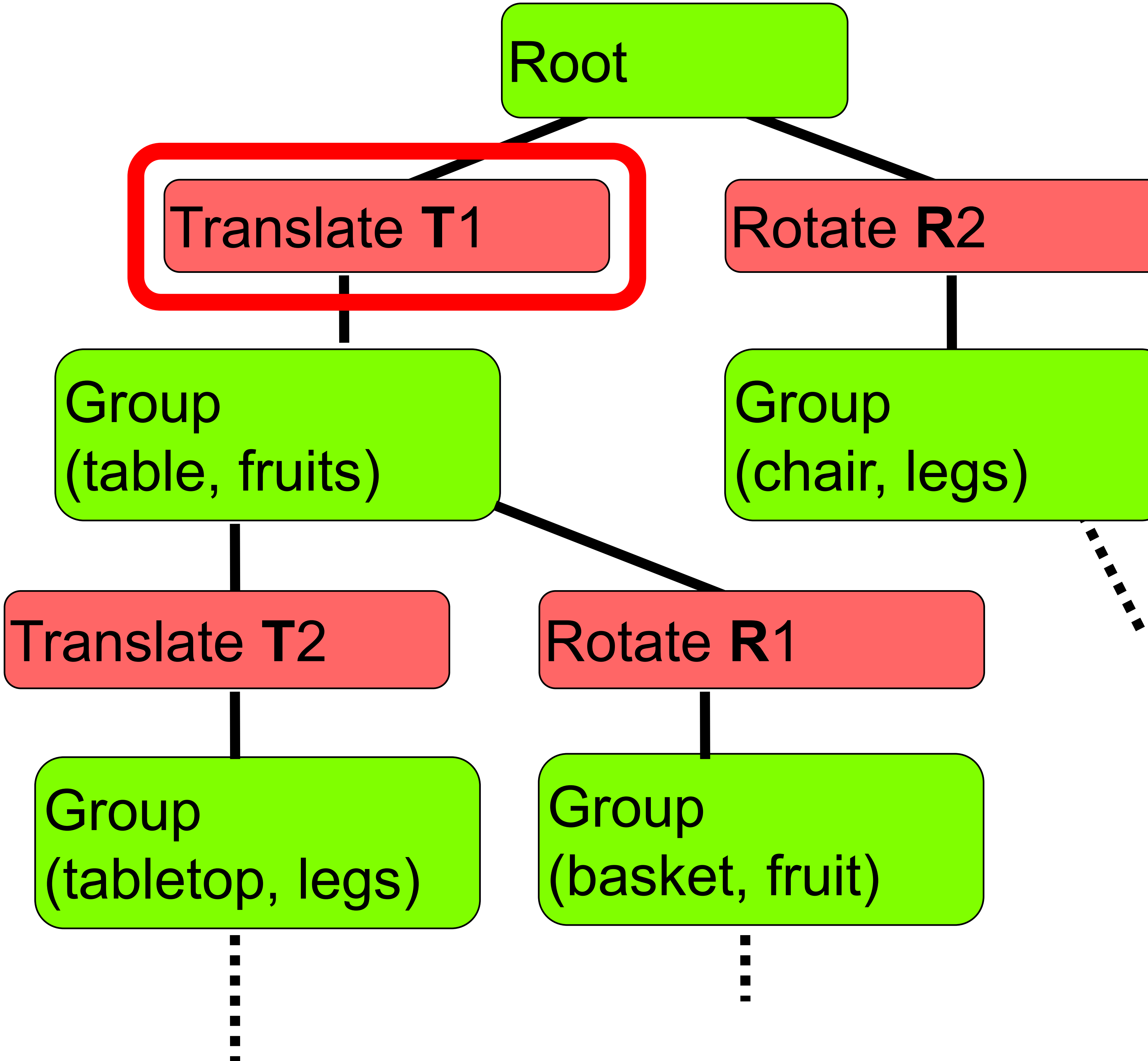
S = **T1 R1**

Traversal Example



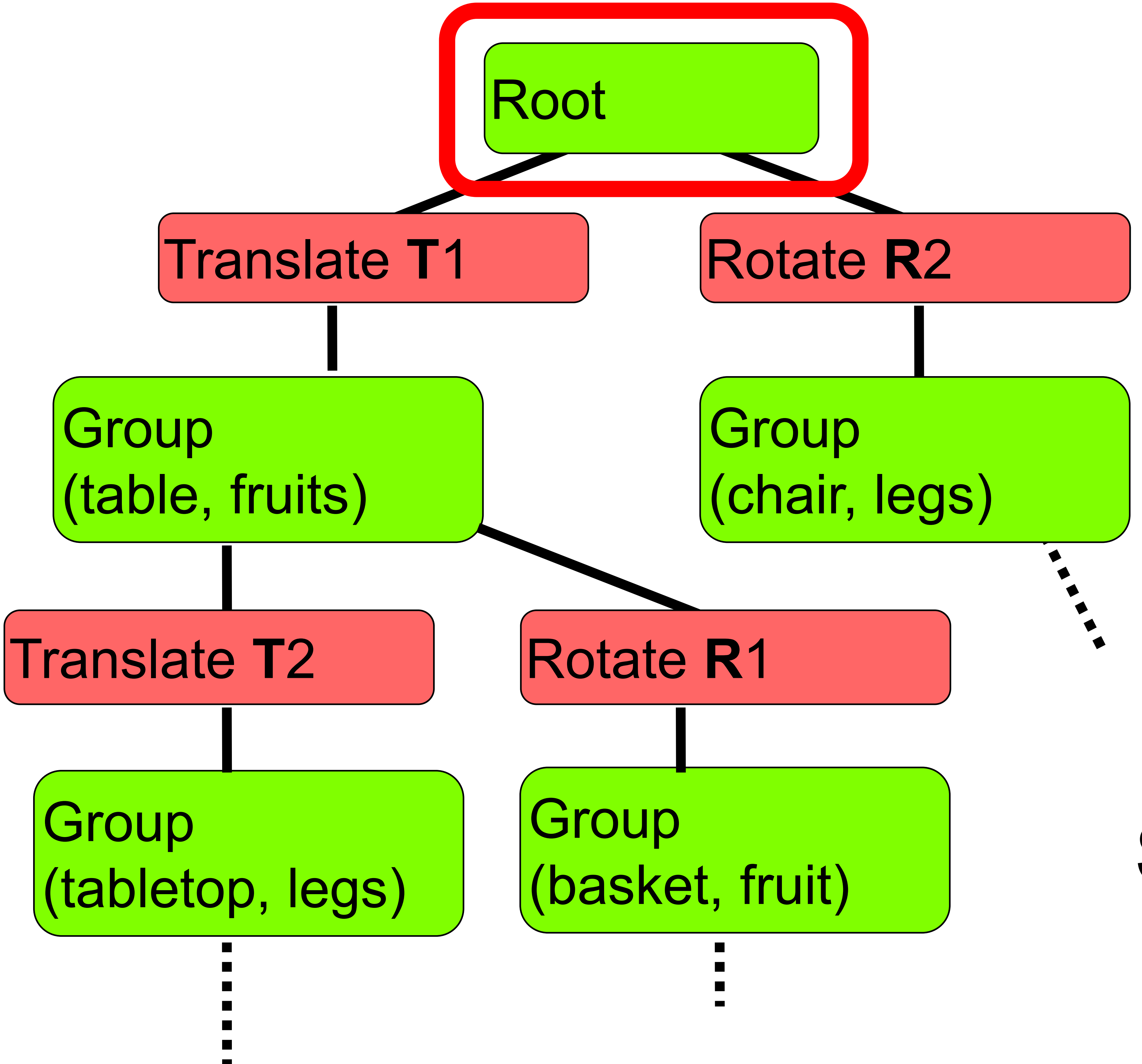
S = **T1**

Traversal Example



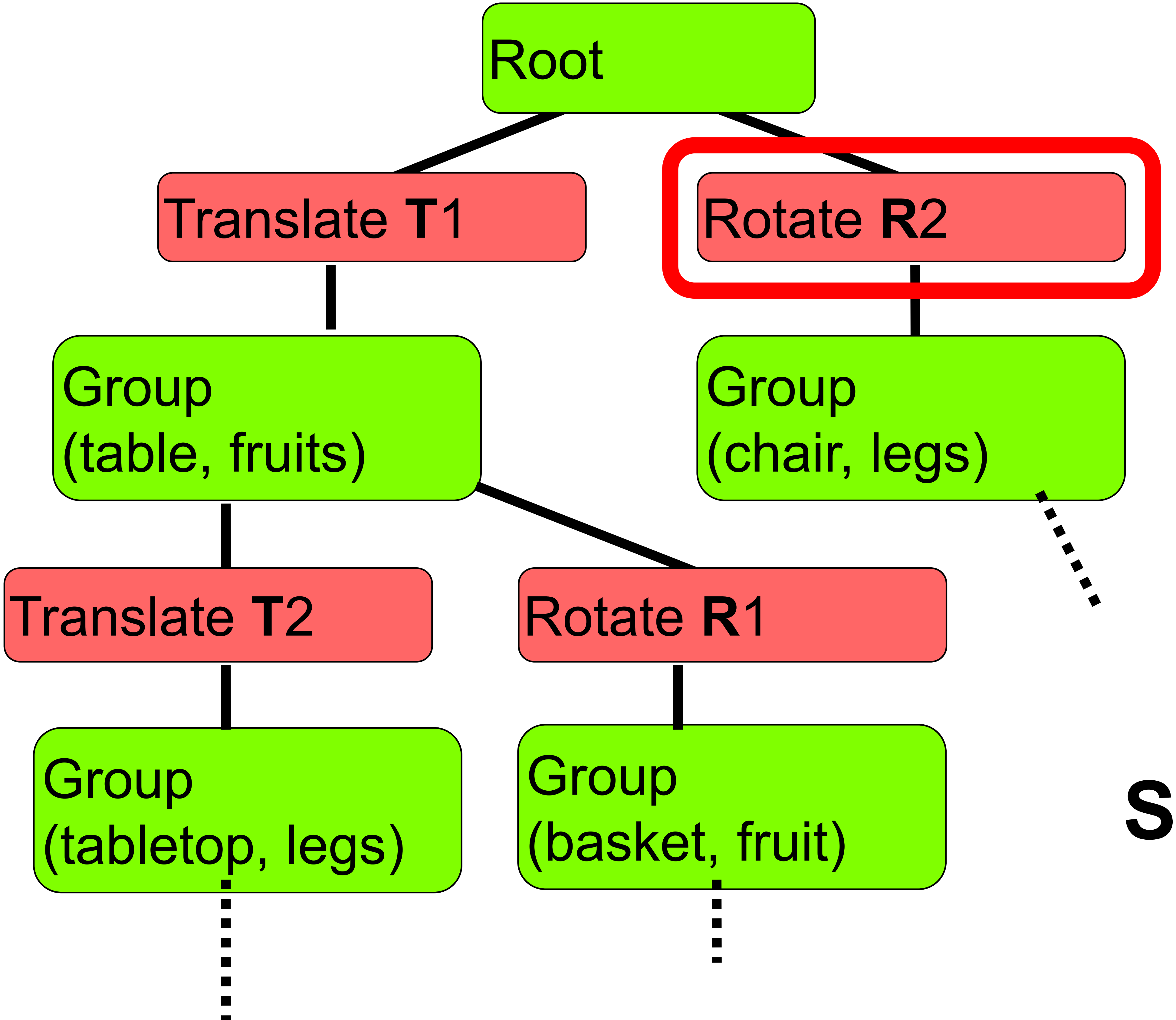
S = **T1**

Traversal Example



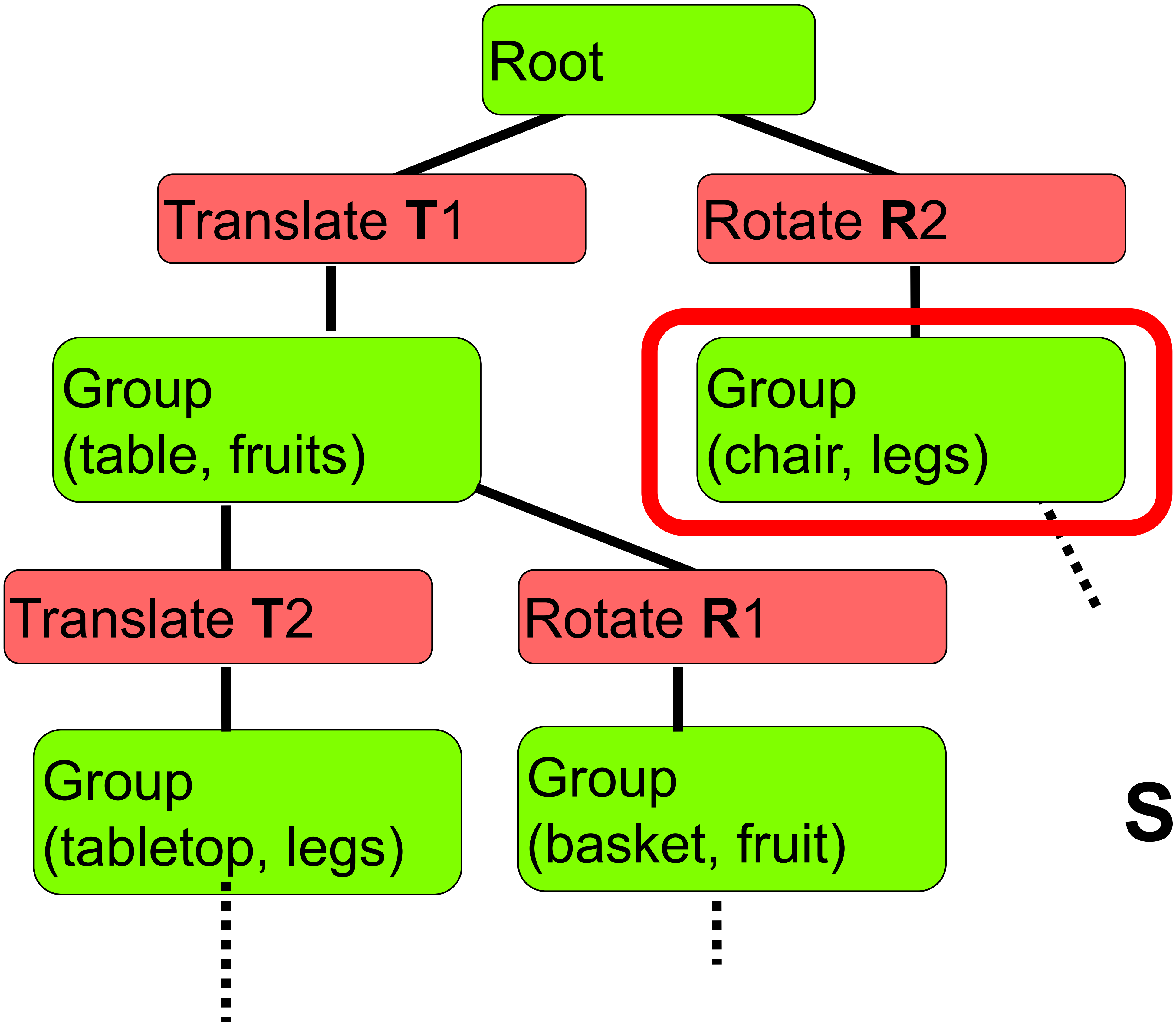
S = **I**

Traversal Example



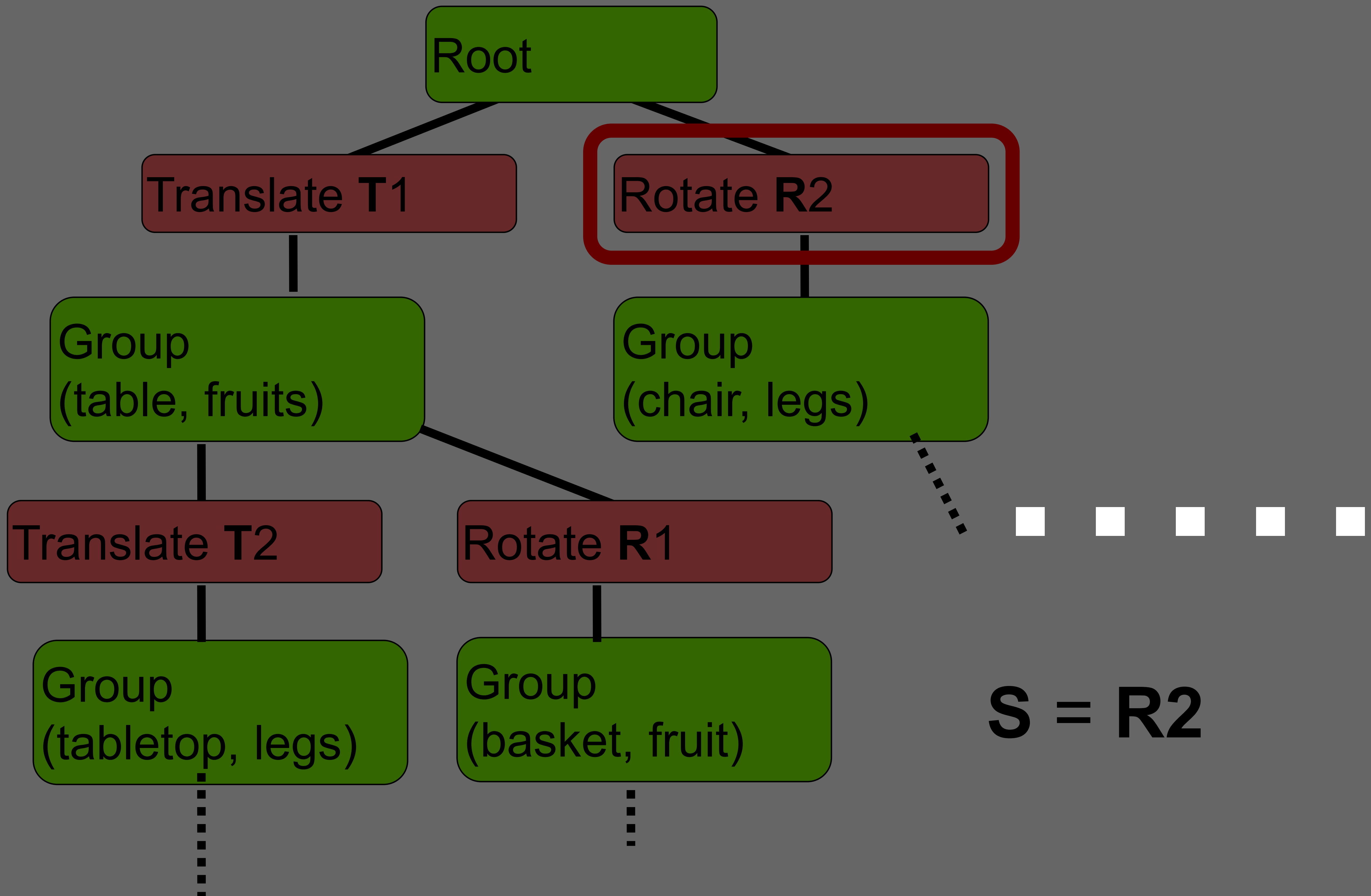
S = R2

Traversal Example

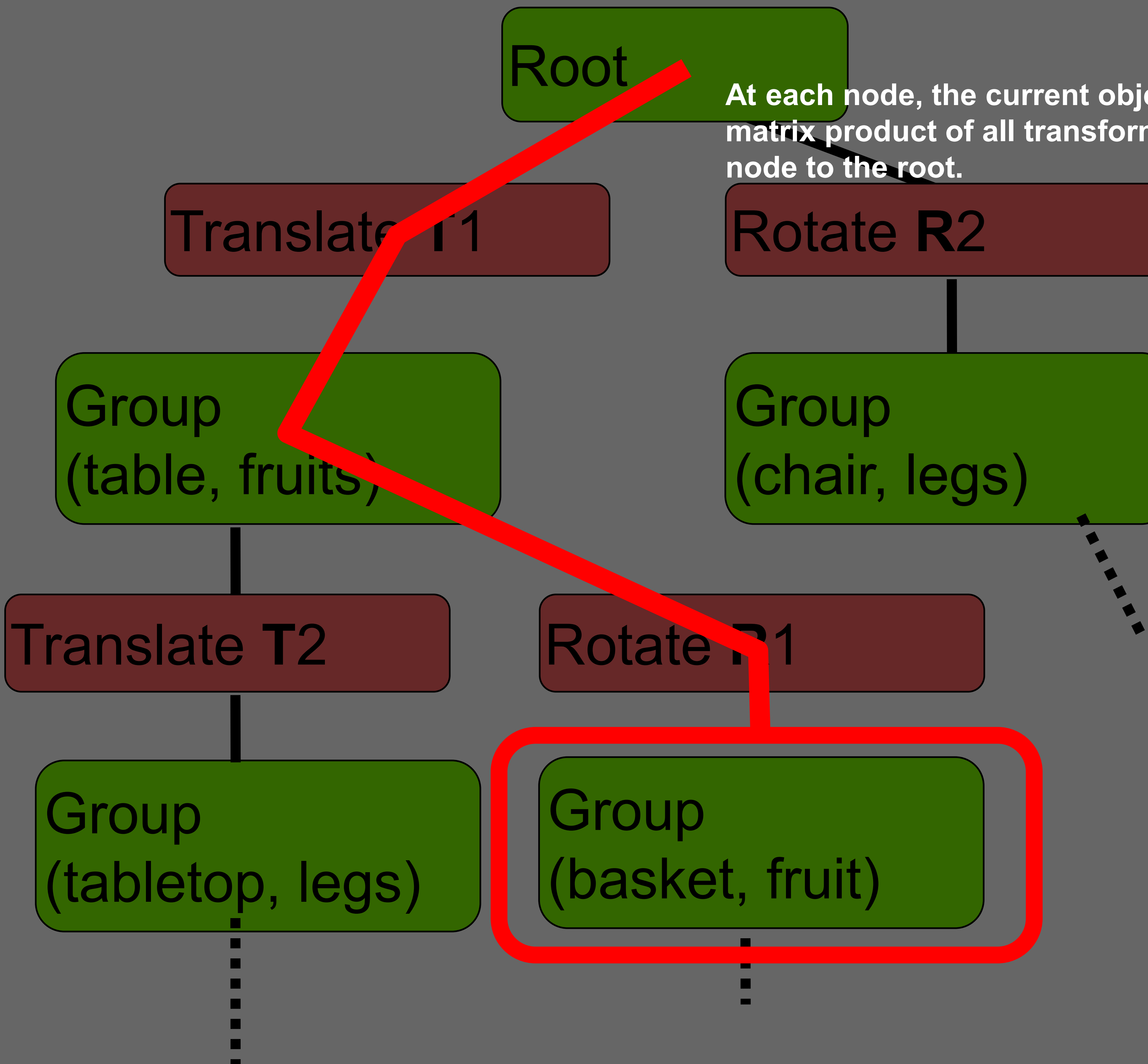


S = **R2**

Traversal Example



Traversal Example



At each node, the current object-to-world transformation is the matrix product of all transformations found on the way from the node to the root.

S = T1R1

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix **(Why?)**

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, "undo" when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? **T***$\mathbf{T}$-1 = **I** does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D object onto a plane – no way to undo, inverse doesn't exist!

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? **T***T-1 = **I** does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D object onto a plane – no way to undo, inverse doesn’t exist!

Can you think of a data structure suited for this?

Traversal State – Stack

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? **T*****T**-1 = **I** does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D object onto a plane – no way to undo, inverse doesn’t exist!

- **Solution:** Keep state variables in a **stack**
  - Push current state when entering node, update current state
  - Pop stack when leaving state-changing node
  - See what the stack looks like in the previous example!