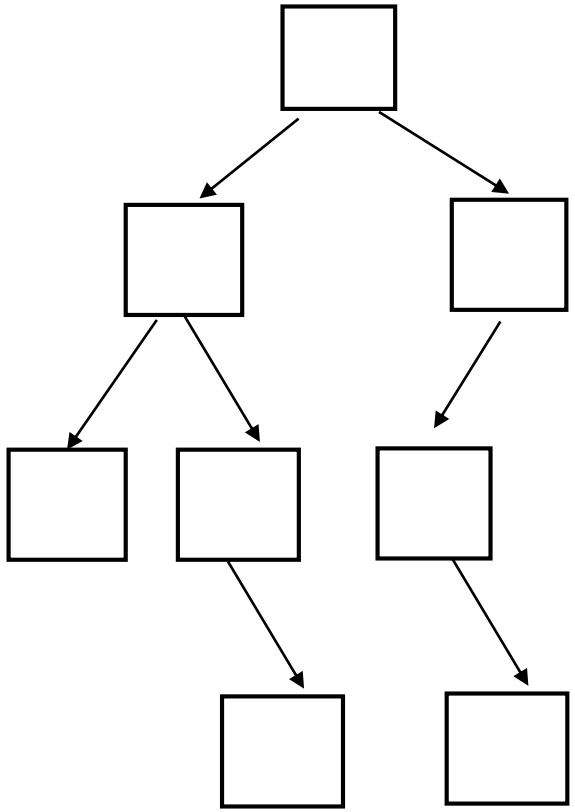
COMP 250

Lecture 19

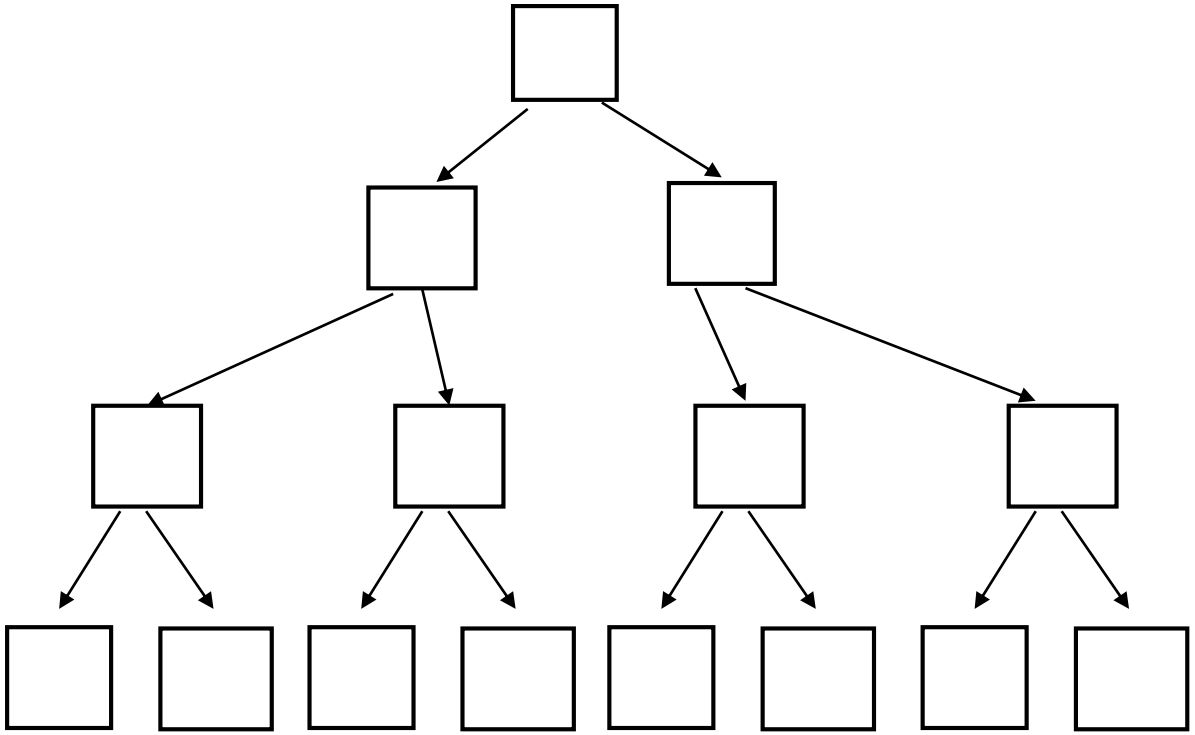
binary trees, expression trees

Oct. 24, 2016

Binary tree:
each node has at most two children.

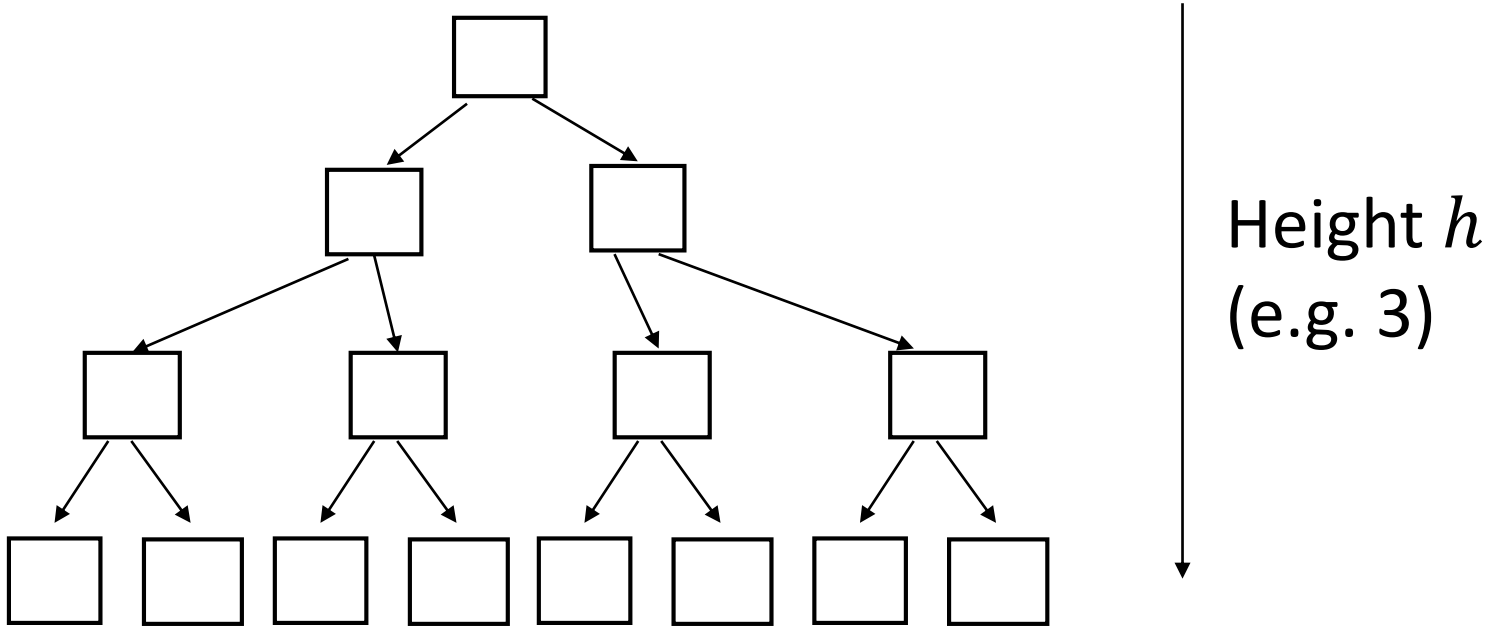


Maximum number of nodes in a binary tree?



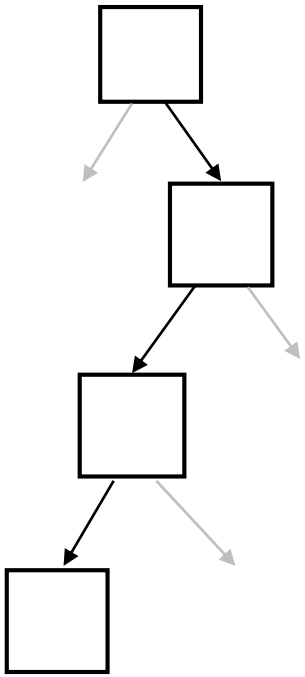
Height h
(e.g. 3)

Maximum number of nodes in a binary tree?



$$n = 1 + 2 + 4 + 8 + 2^h = 2^{h+1} - 1$$

Minimum number of nodes in a binary tree?



Height h
(e.g. 3)

$$n = h + 1$$

```
class  BTree<T>{
    BTNode<T>   root;
       :

    class  BTNode<T>{
       T                  e;
       BTNode<T>    leftchild;
       BTNode<T>    rightchild;
       :
    }
}
```

Binary Tree Traversal (depth first)

Rooted tree
(last lecture)

Binary tree

```
preorder(root){
   if (root is not empty){
        visit root
        for each child of root
              preorder( child )
   }
}
```

Binary Tree Traversal (depth first)

Rooted tree
(last lecture)

```
preorder(root){
   if (root is not empty){
       visit root
       for each child of root
            preorder( child )
   }
}
```

Binary tree

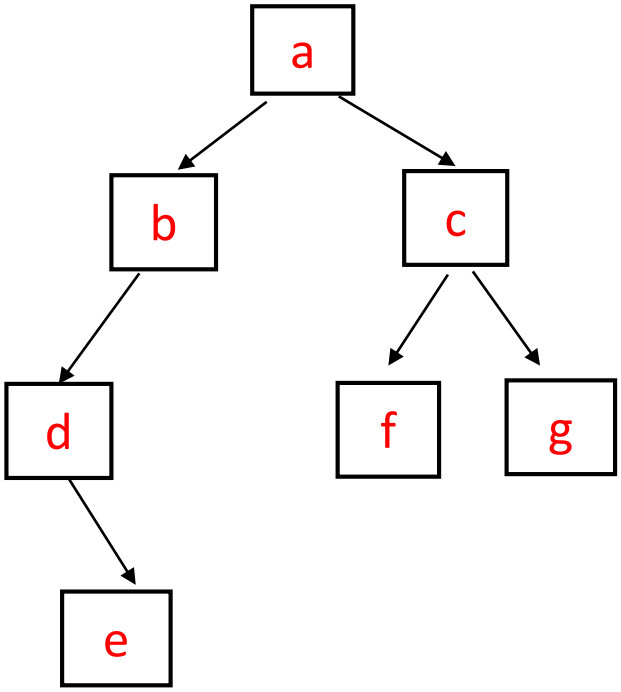
```
preorderBT (root){
   if (root is not empty){
       visit root
       preorderBT( root.left )
       preorderBT( root.right )
   }
}
```

```
preorderBT (root){
   if (root is not empty){
      visit root
      preorderBT( root.left )
      preorderBT( root.right )
  }
}
```

```
postorderBT (root){
   if (root is not empty){
      postorderBT(root.left)
      postorderBT(root.right)
      visit root
   }
}
```

```
inorderBT (root){
   if (root is not empty){
      inorderBT(root.left)
      visit root
      inorderBT(root.right)
  }
}
```

Example

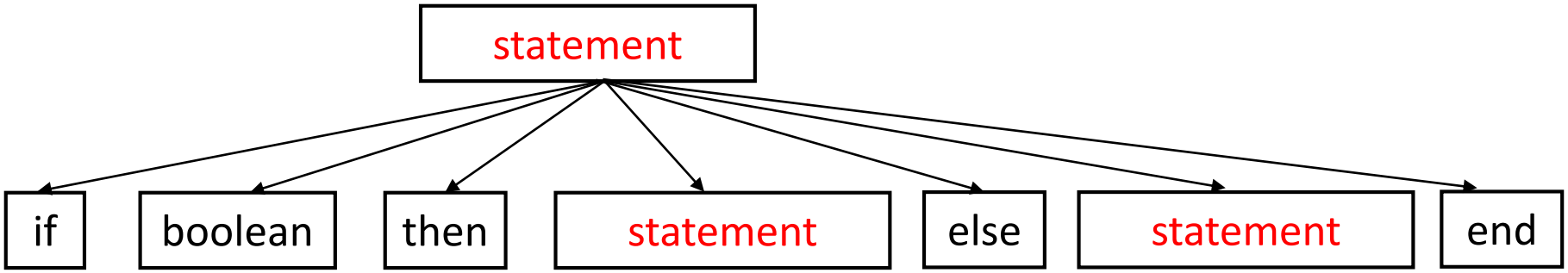


Pre order: abdecfg

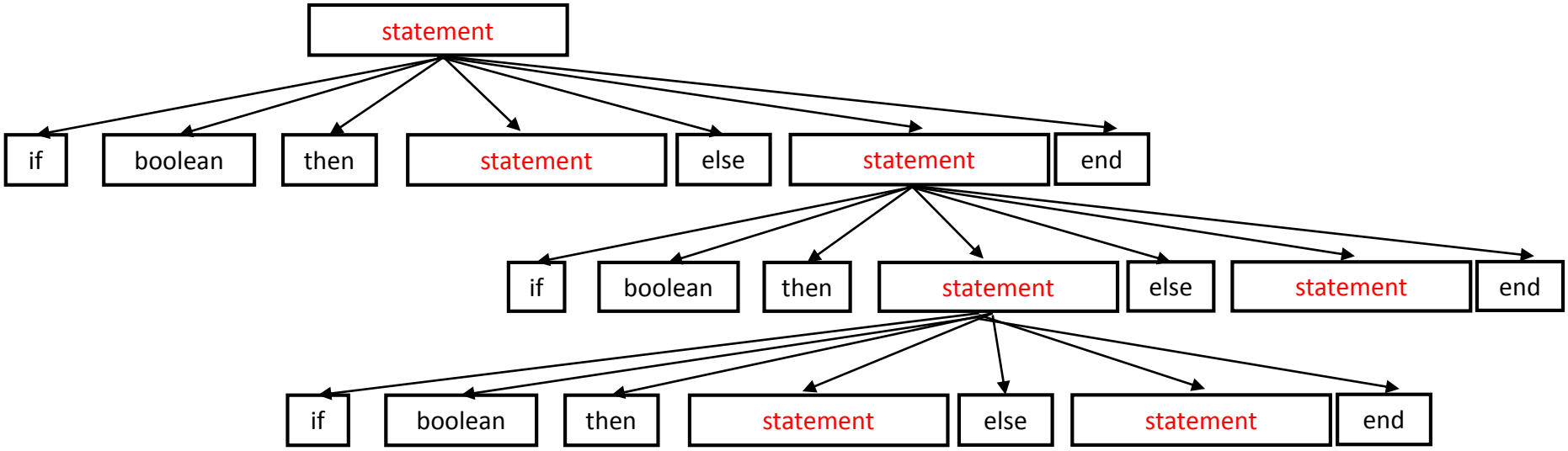
In order: debafcg

Post order: edbfgca

Example of binary tree: "Syntax (sub)Tree" of Assignment 2



statement = if boolean then statement else statement end
| assignment

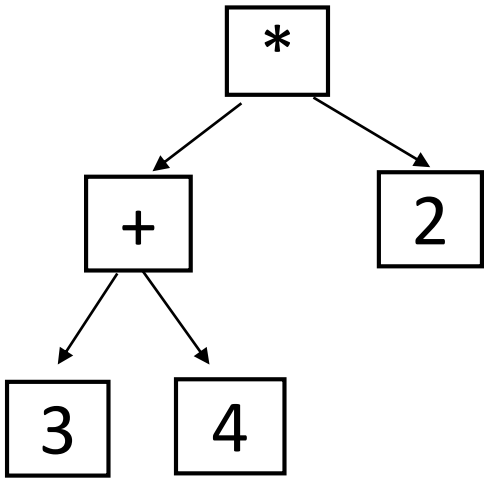


The statement nodes form a binary (sub)tree.

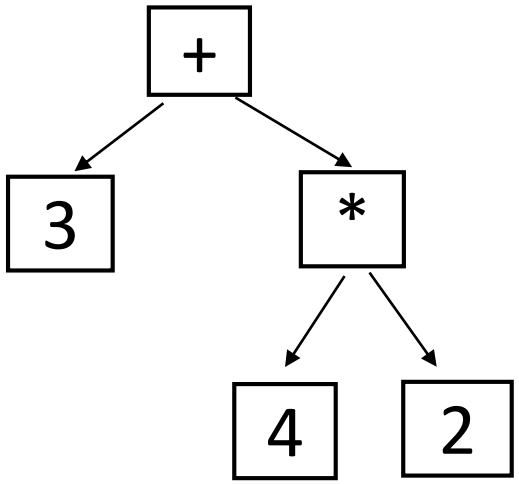
Expression Tree

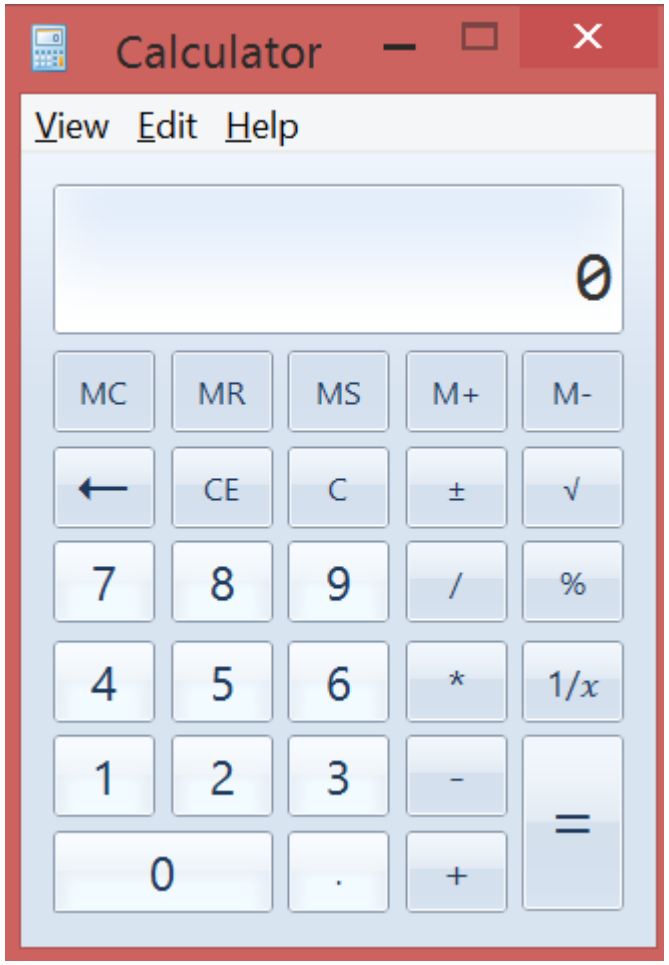
e.g. 3 + 4 * 2

(3 + 4) * 2



3 + (4 * 2)





My Windows calculator says
3 + 4 * 2 = 14.

Why? (3 + 4) * 2 = 14.

Whereas….
if I google “3+4*2”, I get 11.

3 + (4*2) = 11.

Example of expression tree

a – b / c + d * e ^ f ^ g

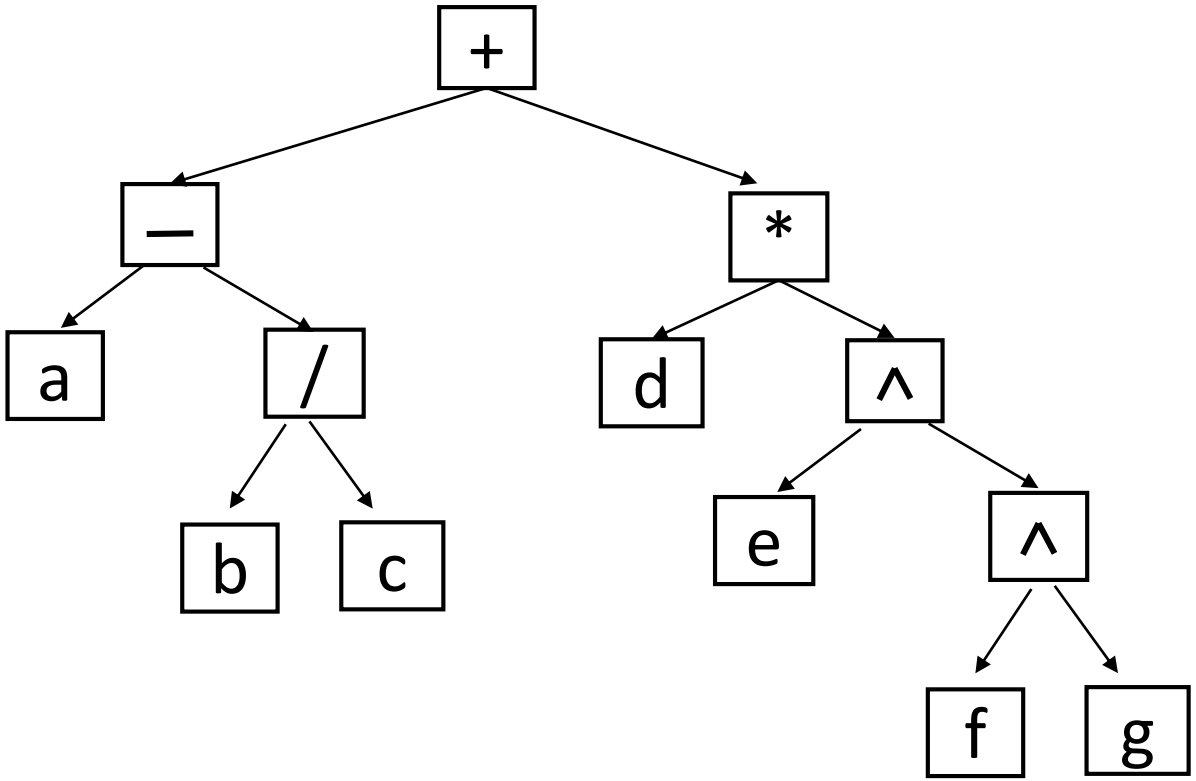
^ is exponentiation

We consider binary operators only
e.g. we don't consider 3 + -4 = 3 + (-4)

Precedence ordering makes brackets unnecessary.
i.e. (a – (b / c)) + (d * (e ^ (f ^ g)))

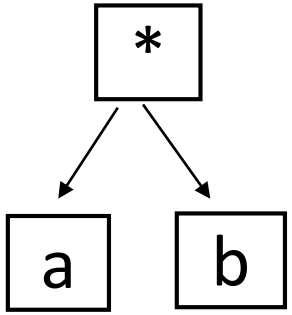
Expression Tree

a – b / c + d * e ^ f ^ g ≡ (a – (b / c)) + (d ∗ (e ^ (f ^ g)))



Internal nodes are operators. Leaves are numbers.

Infix, prefix, postfix expressions



infix: a*b

prefix: *ab

postfix: ab*

Infix, prefix, postfix expressions

baseExp = variable | integer

op = + | - | * | / | ^

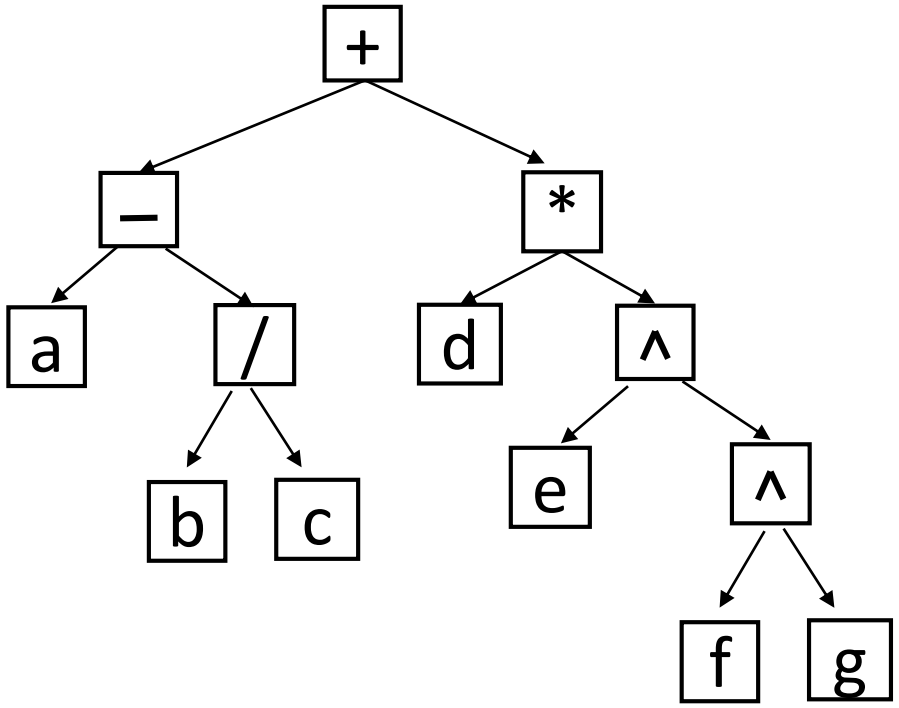
inExp = baseExp | inExp op inExp

preExp = baseExp | op preExp prefExp

postExp = baseExp | postExp postExp op

Use one.

If we traverse an expression tree, in which order do we ‘visit’ nodes ?

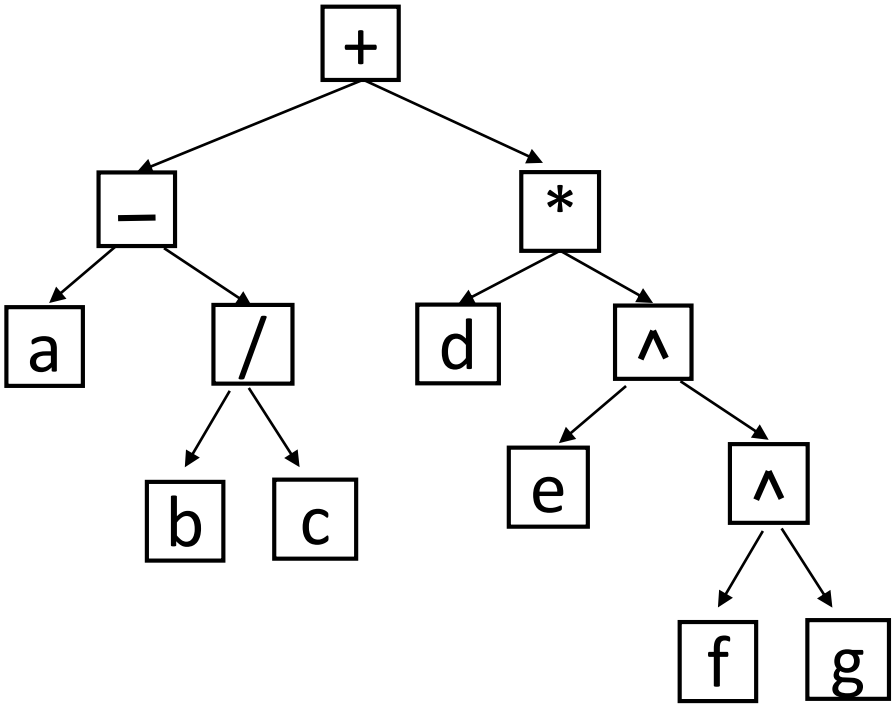


inorder traversal gives infix expression: a – b / c + d * e ^ f ^ g

preorder traversal gives prefix expression: + – a / b c * d ^ e ^ f g

postorder traversal gives postfix expression: a b c / - d e f g ^ ^ * +

If we were given an expression tree, then how would we evaluate the expression ?



If we were given an expression tree, then we could evaluate it using a **postorder traversal**:

```
evalExpTree(root){
    if (root is a leaf)      //  root is a number
          return value
    else{    //  the root is an operator
      firstOperand        =   evalExpTree( root.leftchild )
      secondOperand   =   evalExpTree( root.rightchild )
      return evaluate(firstOperand, root, secondOperand)
    }
}
```

However, in practice we are not given an expression tree.

How to evaluate expressions?

Infix expressions are awkward to evaluate because of precedence ordering.

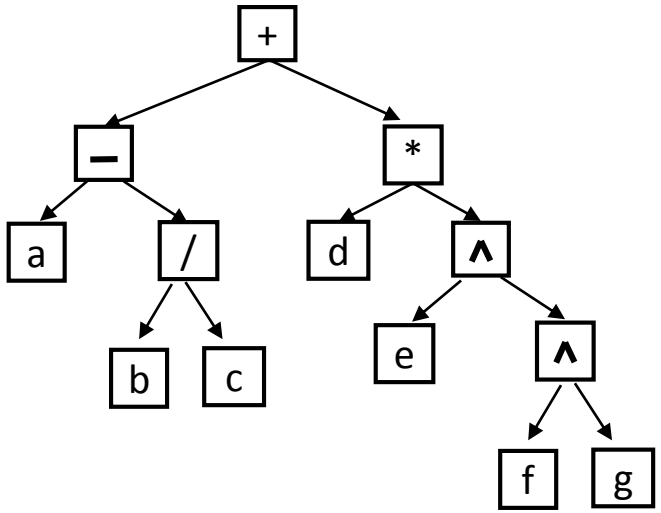
ASIDE: One can convert an infix expression to a postfix expression: http://wcipeg.com/wiki/Shunting_yard_algorithm
Details omitted here. For your interest only.

We next show how to evaluate a postfix expression using a stack.

a b c / - d e f g ^ ^ * +

stack over time

a
ab
abc
a(bc/)
(a(bc/) -)
(a(bc/) -) d
(a(bc/) -) d e
(a(bc/) -) d e f
(a(bc/) -) d e f g
(a(bc/) -) d e (f g ^)
(a(bc/) -) d (e (f g ^) ^)
(a(bc/) -) (d (e (f g ^) ^) *)
((a(bc/) -) (d (e (f g ^) ^) *) +)



Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of elements.

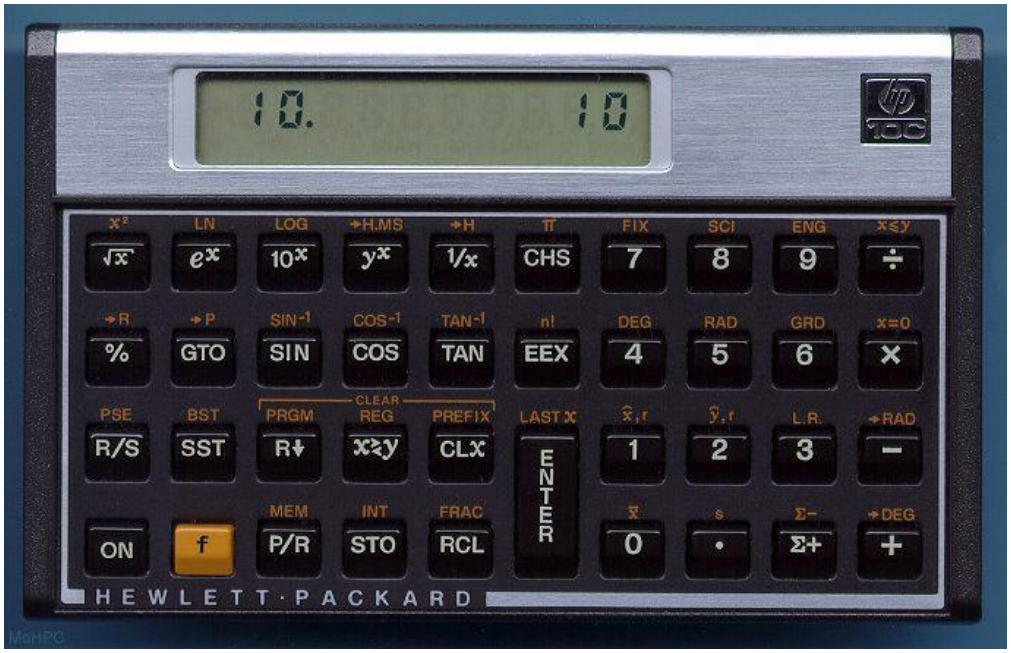
```
s = empty stack
cur = head of expression list
while (cur != null){
    if ( cur.element  is a base expression )
        s.push( cur.element )
    else{                                   //  cur.element is an operator
        operand2  =  s.pop()
        operand1  =  s.pop()
        operator  = cur.element         //   for clarity only
        s.push( evaluate( operand1,  operator,  operand2 )   )
    }
    cur = cur.next
}
```

Prefix expressions called “Polish Notation”

(after Polish logician Jan Lucasewicz 1920’s)

Postfix expressions are called “Reverse Polish notation” (RPN)

Some calculators (esp. Hewlett Packard) require users to input expressions using RPN.



5*4+3 ?

5 <enter>
4 <enter>
* <enter>
3 <enter>
+ <enter>

No “=“ symbol needed on keyboard.