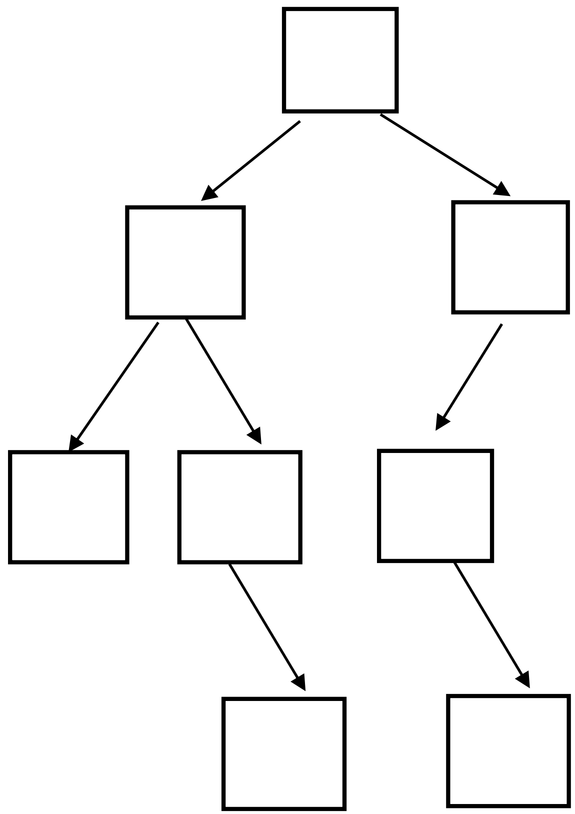
COMP 250

Lecture 24

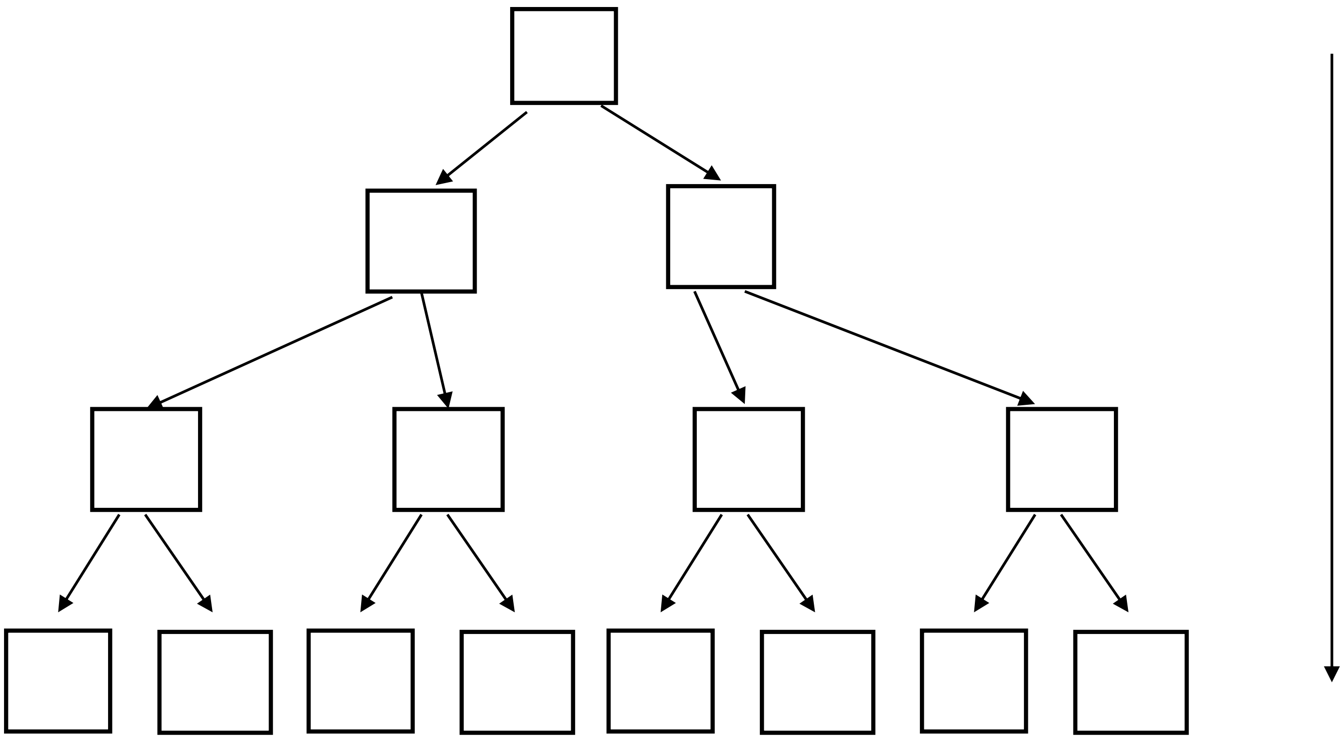
binary trees, expression trees

Nov. 5, 2018

Binary tree:
each node has *at most* two children.

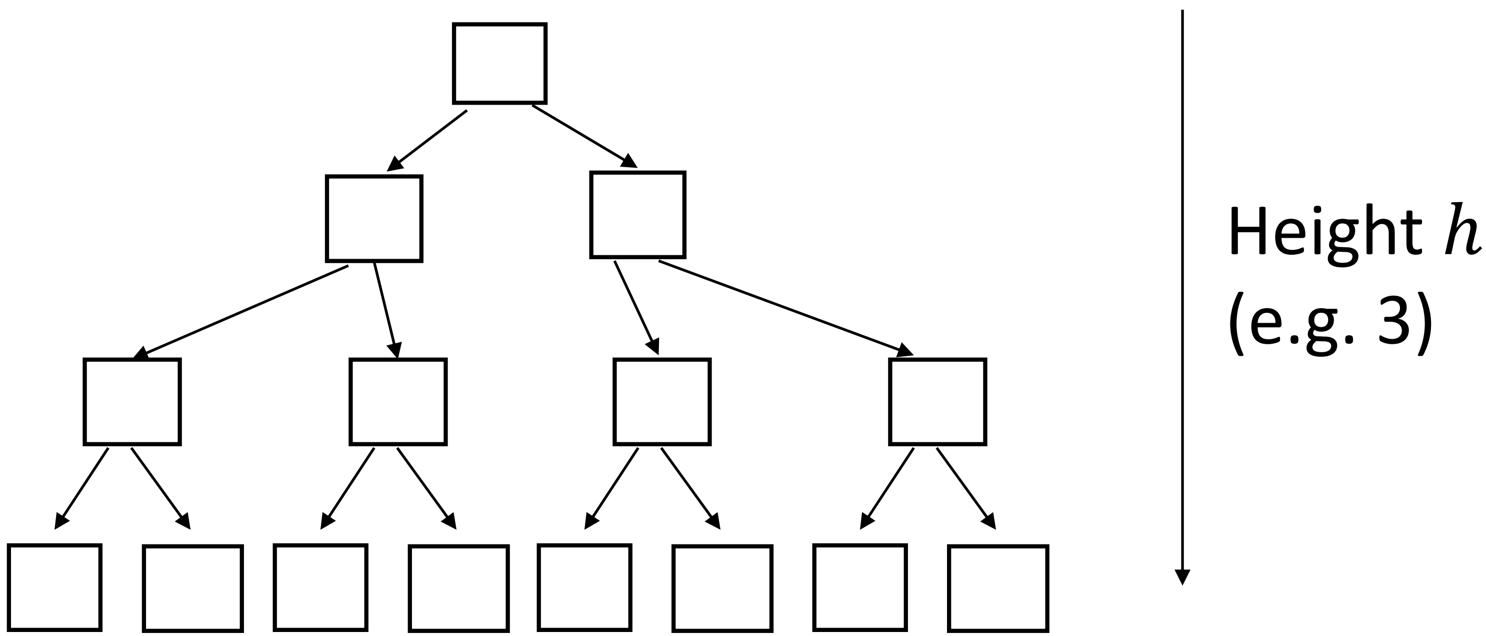


Maximum number of nodes in a binary tree?



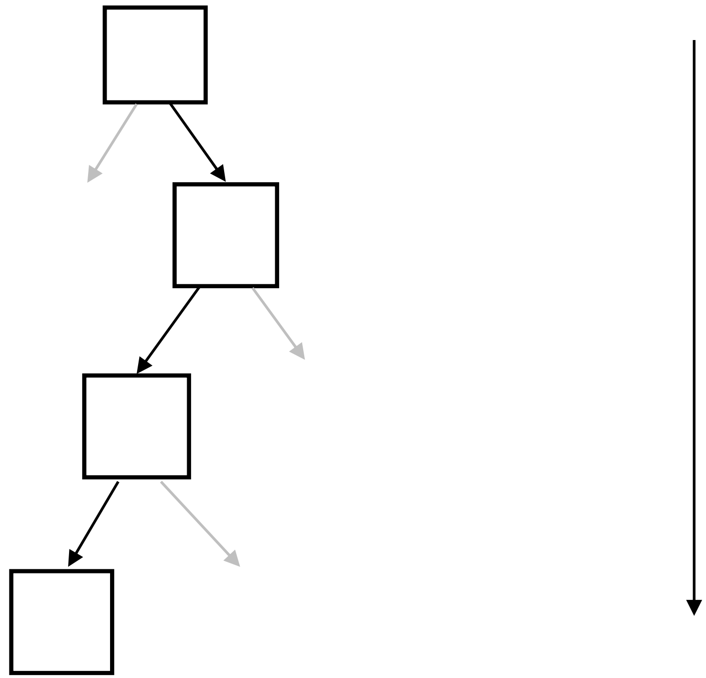
Height h
(e.g. 3)

Maximum number of nodes in a binary tree?



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

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;
       :
    }
}
```

Recall last lecture

// pre-order

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

// post-order

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

```
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
   }
}
```

```
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
  }
}
```

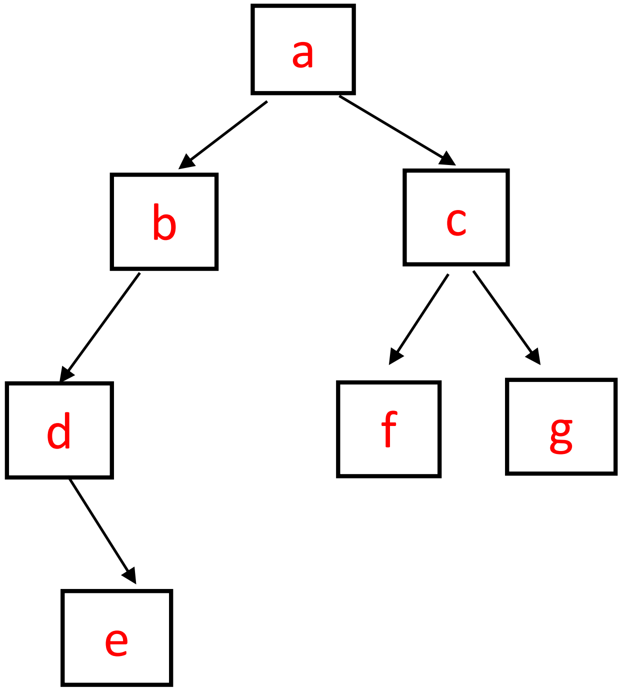
```
inorderBT (root){

}
```

```
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

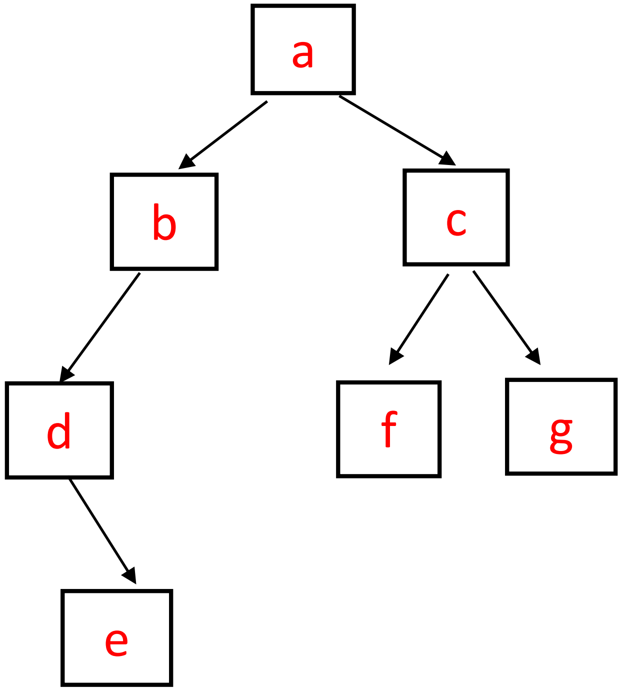


Pre order:

In order:

Post order:

Example

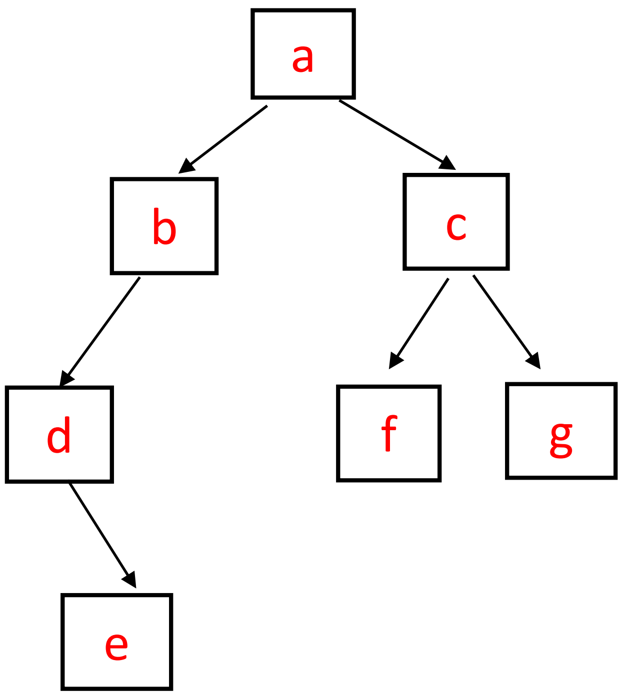


Pre order: a b d e c f g

In order:

Post order:

Example

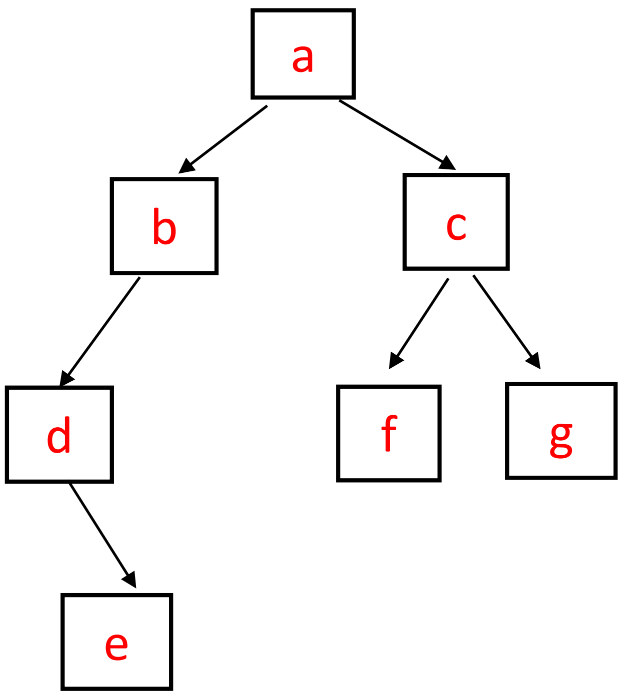


Pre order: a b d e c f g

In order: d e b a f c g

Post order:

Example



Pre order: a b d e c f g

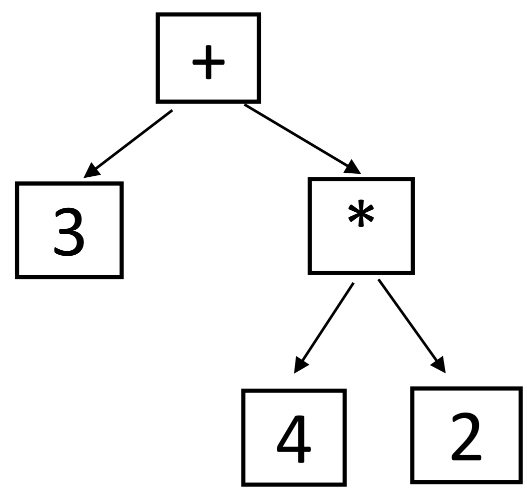
In order: d e b a f c g

Post order: e d b f g c a

Expression Tree

e.g.   3 + 4 * 2

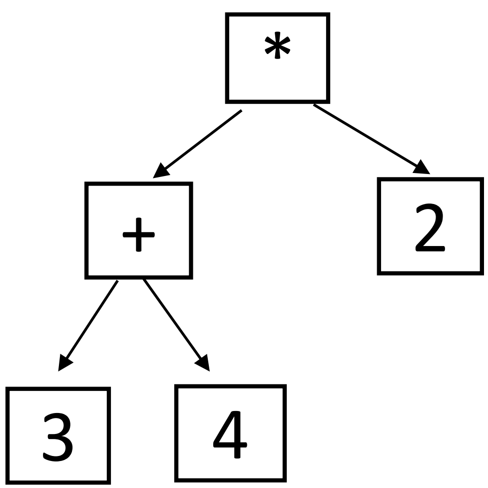
3 + (4 * 2)



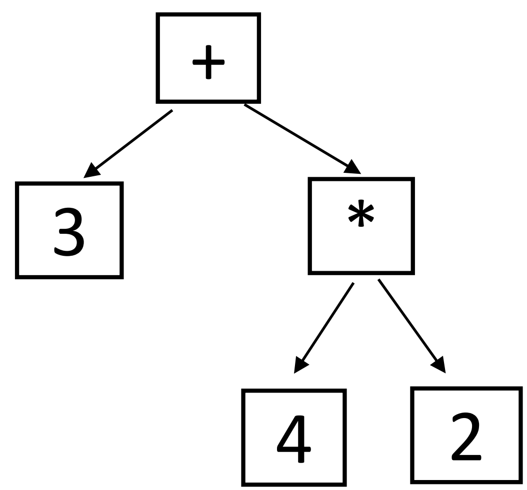
Expression Tree

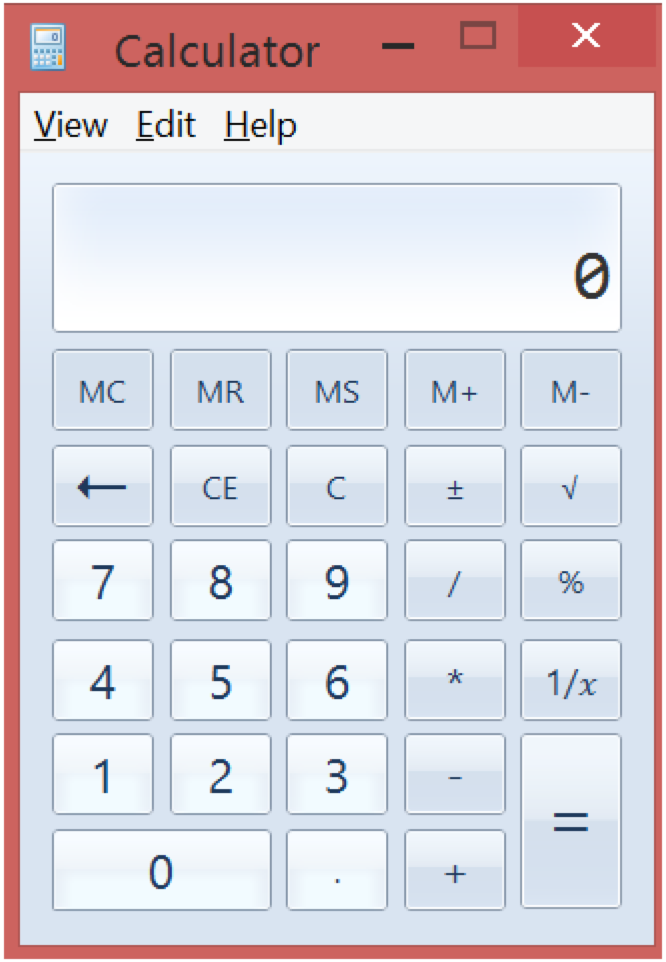
e.g. 3 + 4 * 2

(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.

We can make expressions using binary operators +, -, *, /, ^

e.g. $a - b / c + d * e \wedge f \wedge g$

We can make expressions using binary operators +, -, *, /, ^

e.g. a – b / c + d * e ^ f ^ g

^ is exponentiation: e ^ f ^ g = e ^ (f ^ g)

Operator precedence ordering makes brackets unnecessary.

(a – (b / c)) + (d * (e ^ (f ^ g)))

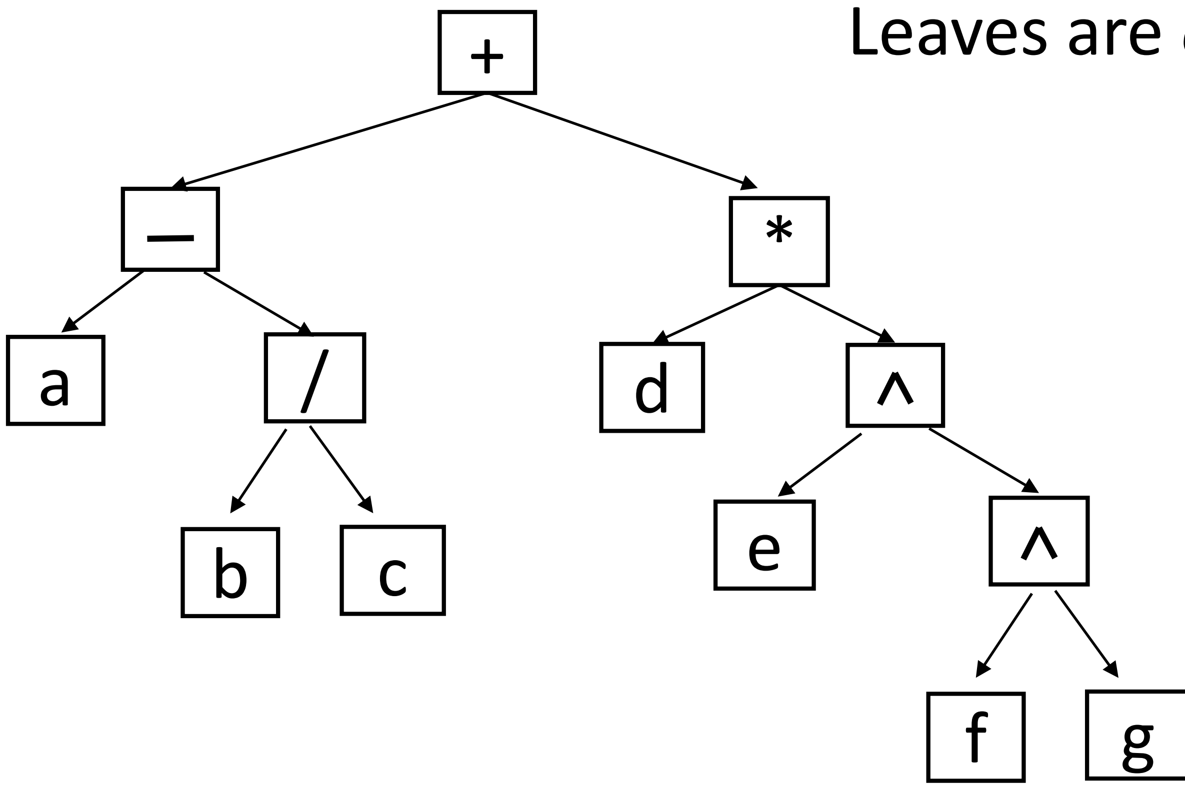
We don’t consider unary operators e.g. 3 + -4 = 3 + (-4)

Expression Tree

$$a - b / c + d * e \wedge f \wedge g \equiv (a - (b / c)) + (d * (e \wedge (f \wedge g)))$$

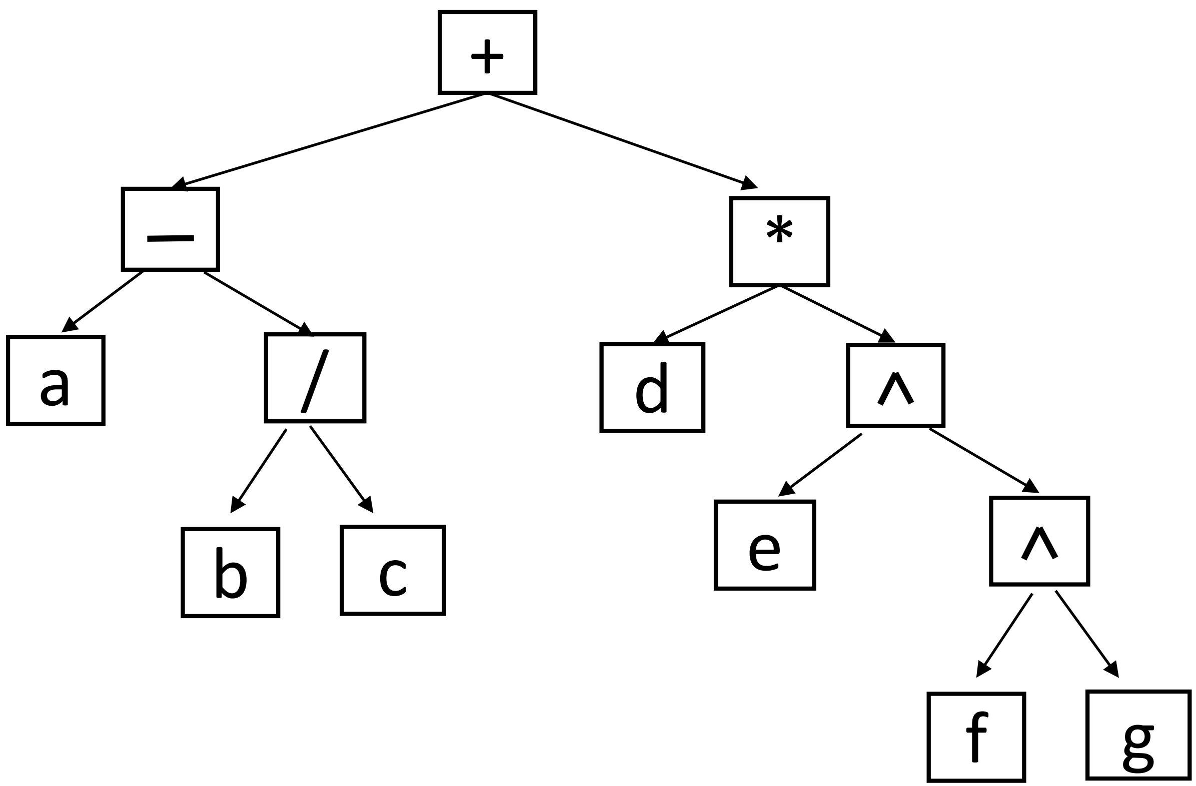
Internal nodes are *operators*.

Leaves are *operands*.

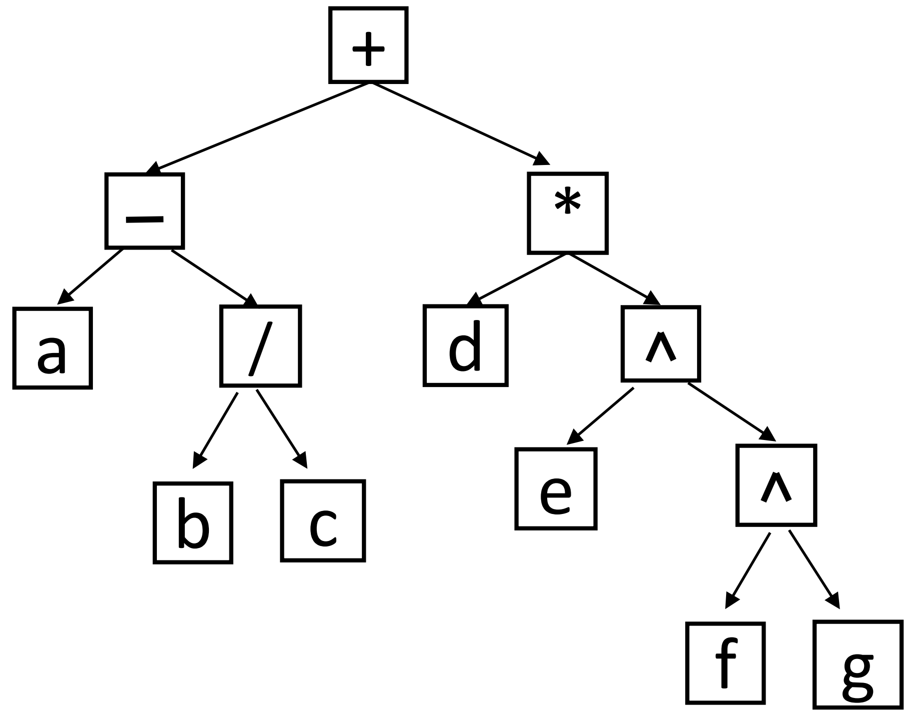


An expression tree can be a way of *thinking about* the ordering of operations used when evaluating an expression.

But to be concrete, *let's say we have a binary tree data structure*:

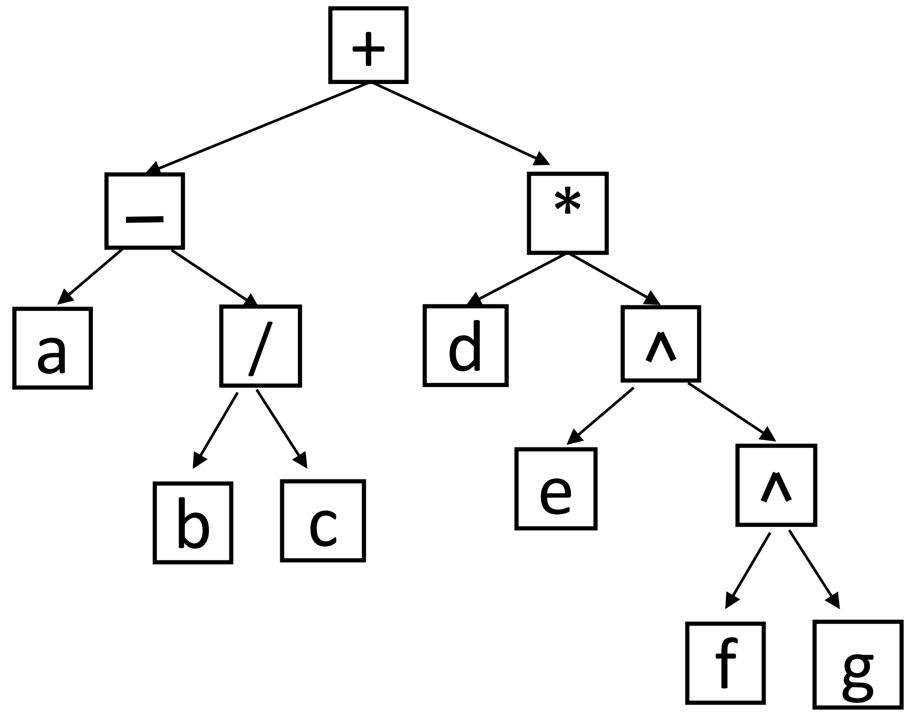


If we traverse an expression tree, and *print out* the node label, what is the expression printed out?



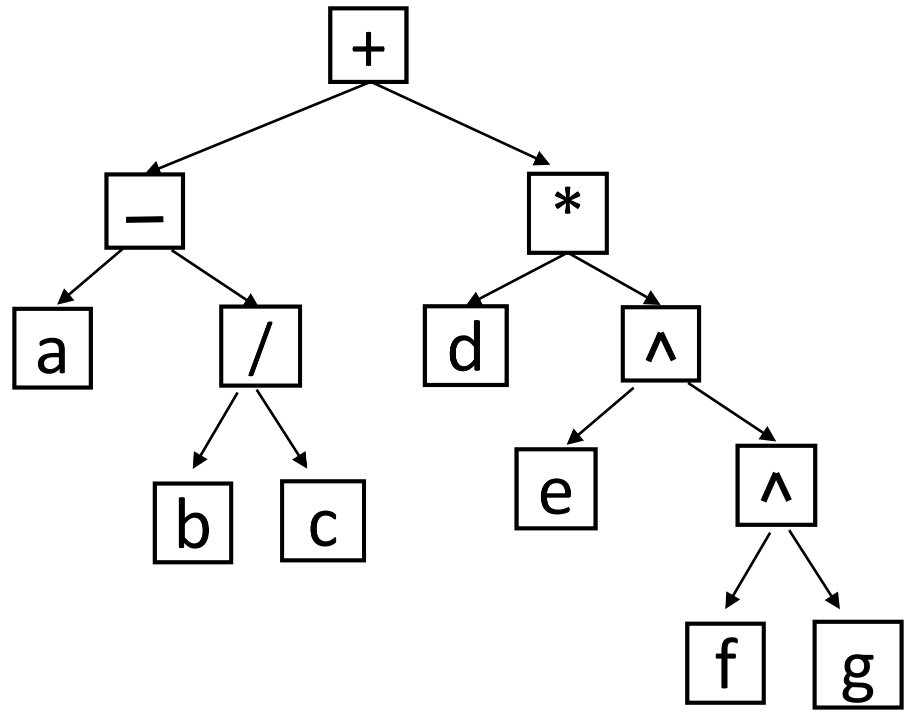
preorder traversal gives :

If we traverse an expression tree, and *print out* the node label, what is the expression printed out?



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

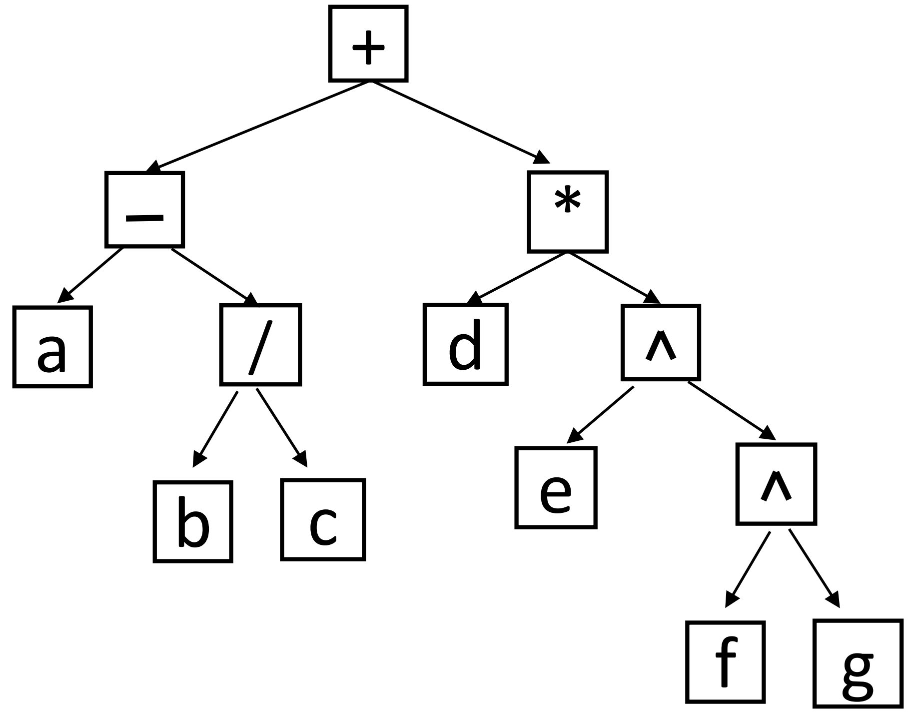
If we traverse an expression tree, and print out the node label, what is the expression printed out?



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

inorder traversal gives :

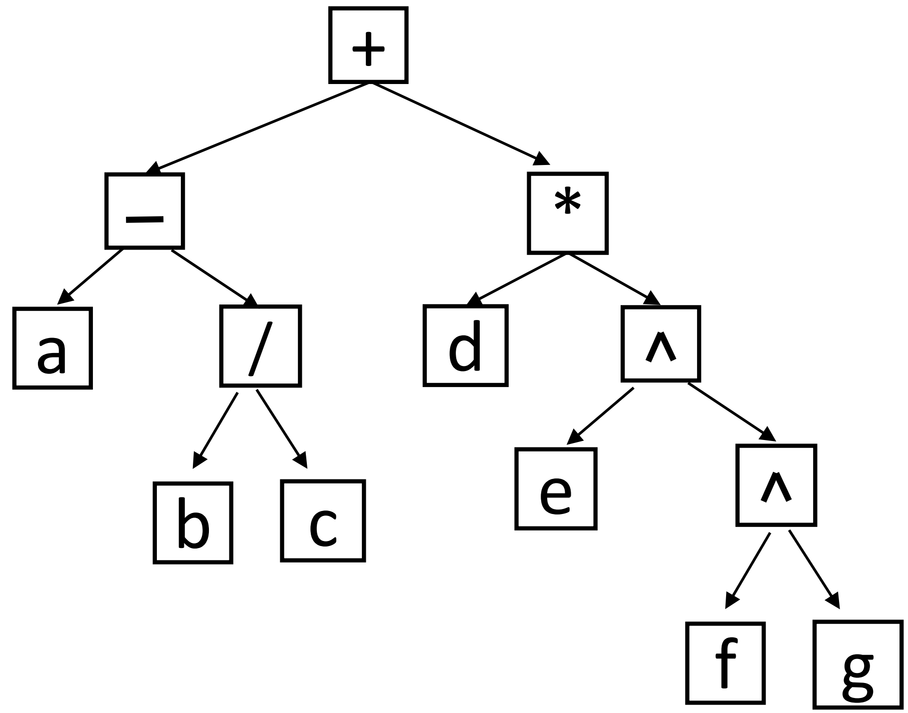
If we traverse an expression tree, and print out the node label, what is the expression printed out?



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

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

If we traverse an expression tree, and print out the node label, what is the expression printed out?

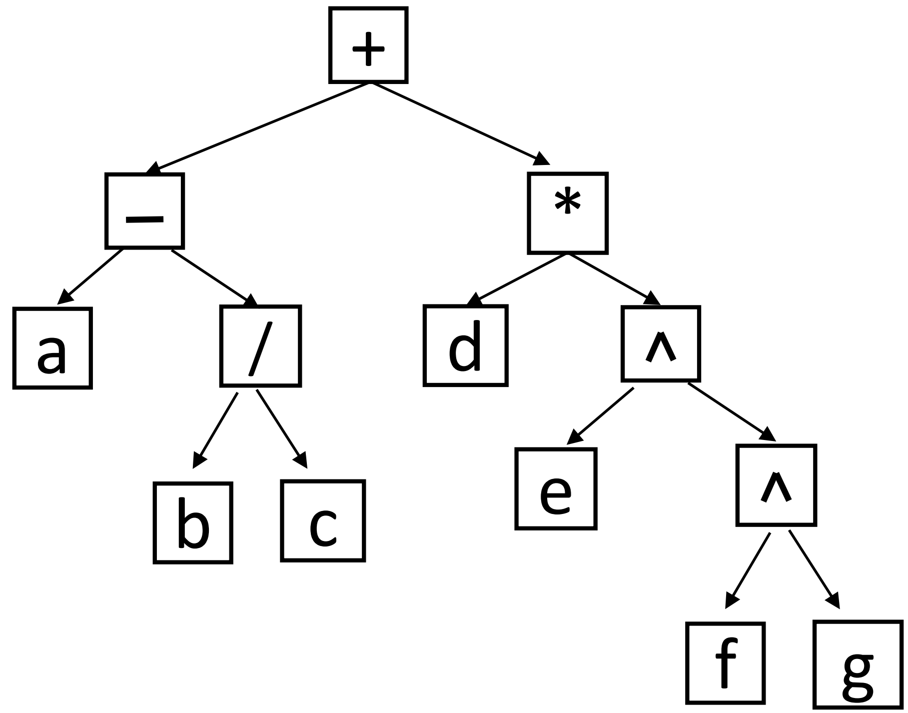


preorder traversal gives : + – a / b c * d ^ e ^ f g

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

postorder traversal gives :

If we traverse an expression tree, and print out the node label, what is the expression printed out?

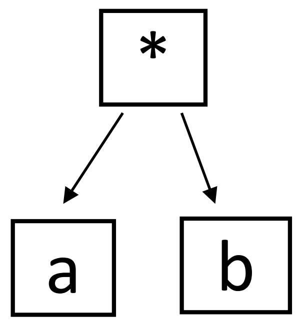


preorder traversal gives : + – a / b c * d ^ e ^ f g

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

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

Prefix, infix, postfix expressions



prefix: * a b

infix: a * b

postfix: a b *

Prefix expressions

baseExp = a | b | c | d .... etc

op = + | - | * | / | ^

preExp = baseExp | op preExp preExp

where | means 'or'

Prefix, infix, postfix expressions

baseExp = a | b | c | d .... etc

op = + | - | * | / | ^

preExp = baseExp | op preExp prefExp

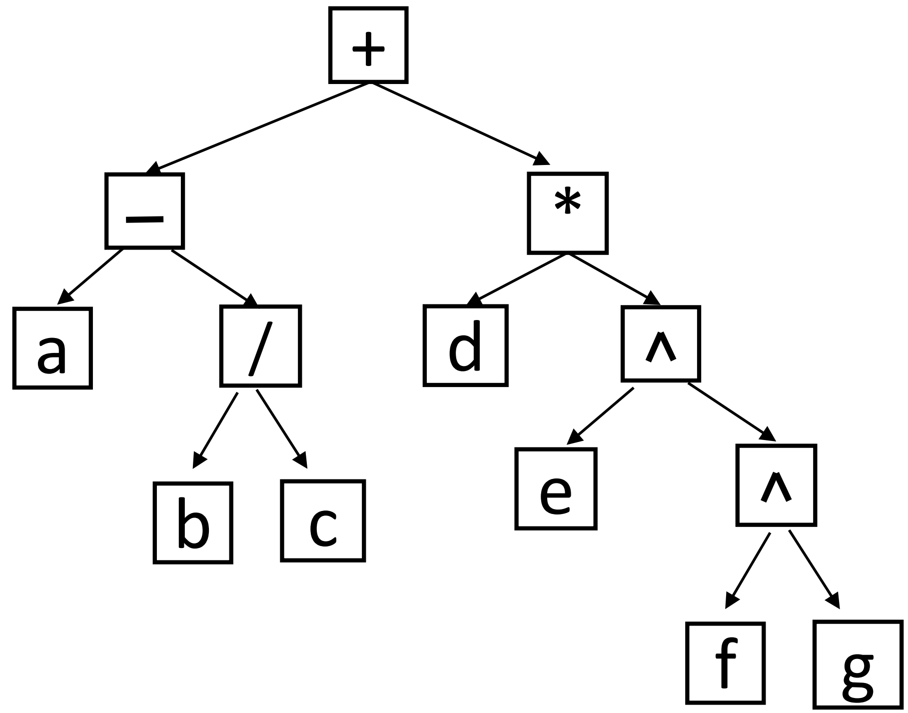
inExp = baseExp | inExp op inExp

postExp = baseExp | postExp postExp op

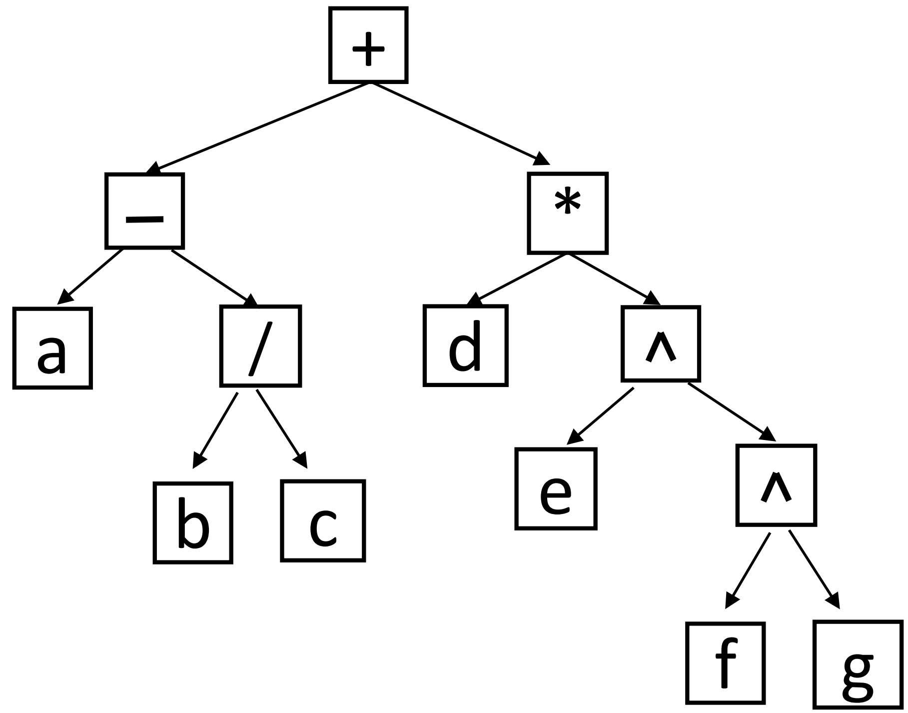
Use only one.

If we traverse an expression tree, and print out the node label, what is the expression printed out?

(same question as four slides ago)



If we traverse an expression tree, and print out the node label, what is the expression printed out? (same question as four slides ago)



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

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

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

Prefix expressions called "Polish Notation"

(after Polish logician Jan Lucasewicz 1920's)

Postfix expressions are called "Reverse Polish notation" (RPN)

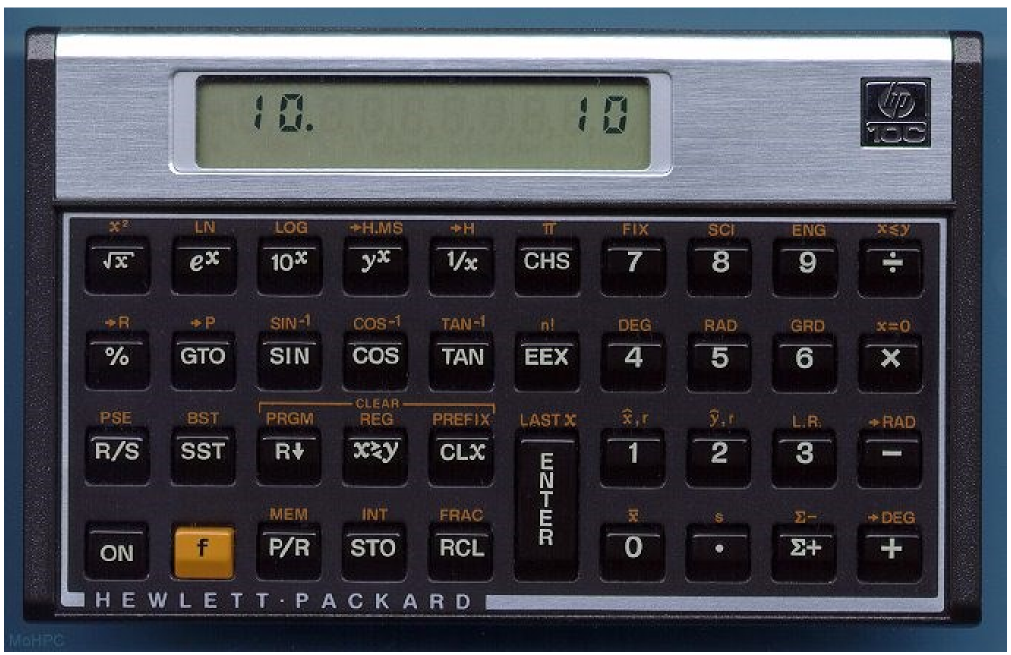
Some calculators (esp. Hewlett Packard) require users to input expressions using RPN.

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.



Calculate 5 * 4 + 3 :

5 <enter>

4 <enter>

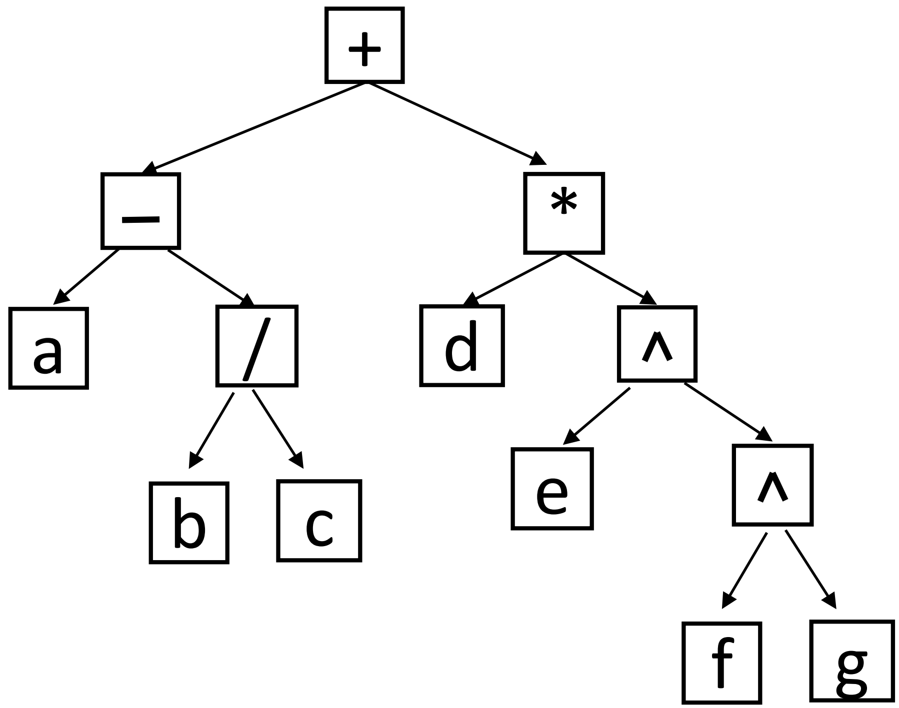
* <enter> → yields 20

3 <enter>

+ <enter> → yields 23

No “=“ symbol on keyboard.

Suppose we are given an expression tree.
How can we evaluate the expression ?



We use a **postorder traversal** (recursive algorithm):

```
evalExpTree(root){
    if (root is a leaf)       //  root is a number
        return value
    else{                     //  root is an operator

                 ?

    }
}
```

We use a **postorder traversal** (recursive algorithm):

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

Postfix expressions *without brackets* are easy to evaluate.

Use one stack, namely for values (not operators).

A similar algorithm applies for pre-fix expressions.

Infix expressions are much more difficult to evaluate.

SLIDE ADDED AFTER CLASS

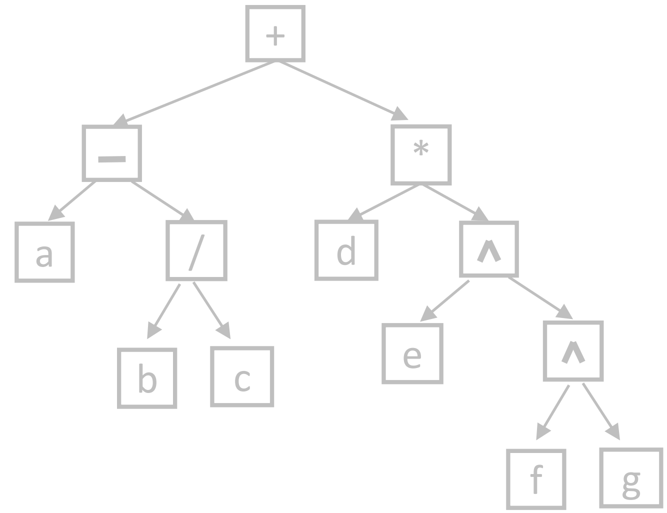
After class, a student asked me why we are learning this. In a nutshell, here is my answer: when you write an expression in Java, you use infix. The computer needs to evaluate that expression. How does it do so? The answer is that the *compiler* converts the infix expression into a postfix (or prefix) expression. Then, when the computer later *runs* the program, the program uses an algorithm such as what I explain next to evaluate this postfix expression. So this is real stuff, happening with every program you write which has expressions in it.

Example:

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

a

stack
over
time



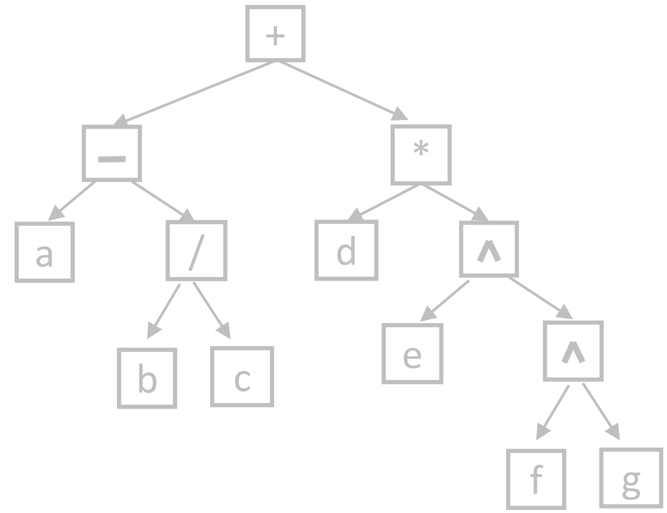
This expression tree is not given. It is shown here so that you can visualize the expression more easily.

Example:

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

a
a b
a b c

stack
over
time



This expression tree is not given. It is shown here so that you can visualize the expression more easily.

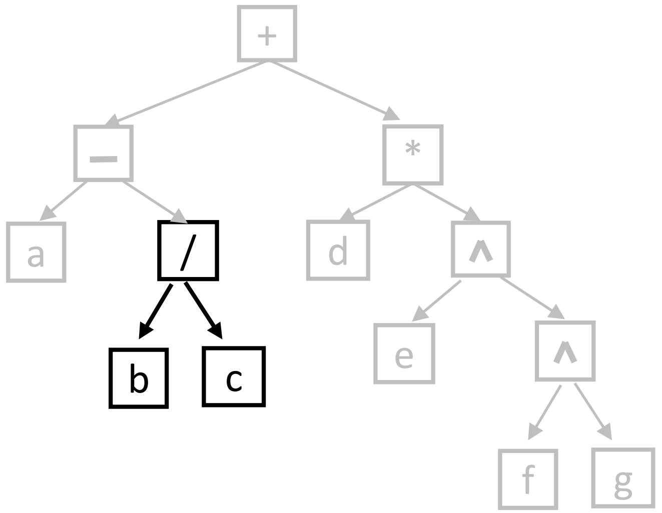
a b c / - d e f g ^ ^ * +

a

a b

a b c

a (b c /)



stack
over
time

We don’t push operator onto stack.

Instead we pop value twice, evaluate, and push.

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

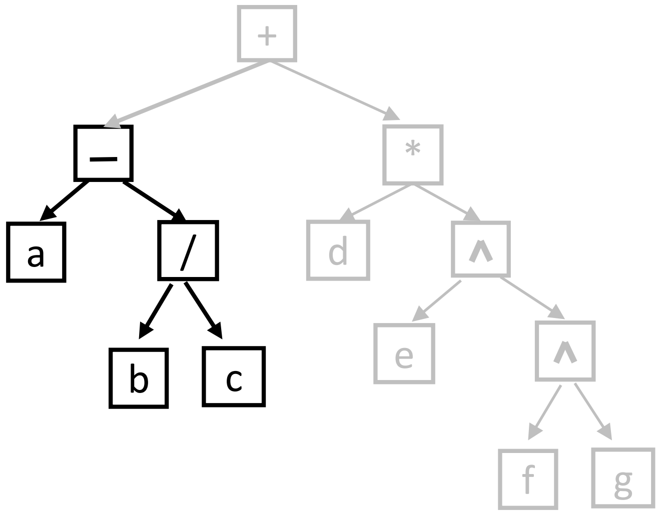
a

a b

a b c

a (b c /)

(a (b c /) -)



stack

over

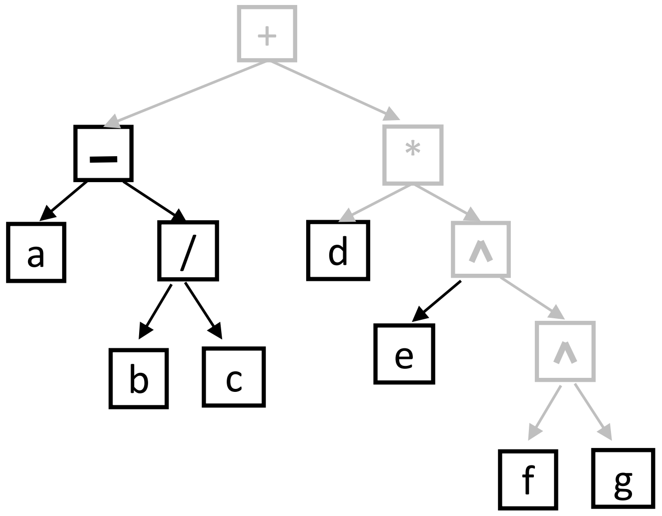
time

Now there is one
value on the stack.

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

stack
over
time

a
a b
a b c
a (b c /)
(a (b c /) -)
:
(a (b c /) -) d e f g



Now there are five values on the stack.

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

stack over time

a

a b

a b c

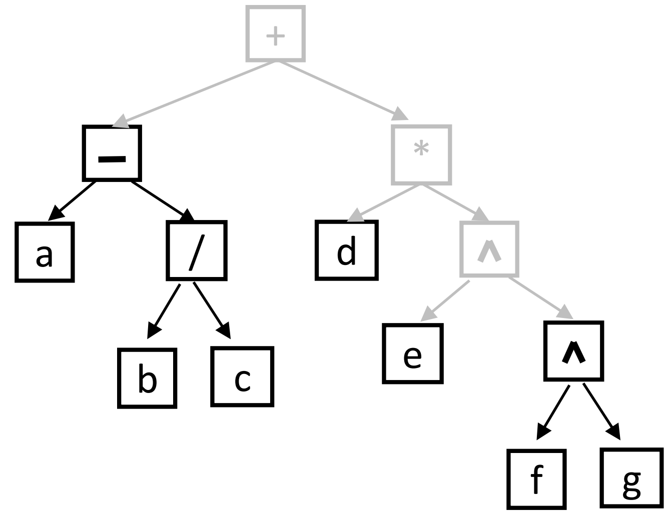
a (b c /)

(a (b c /) -)

:

(a (b c /) -) d e f g

(a (b c /) -) d e (f g ^)



Now there are four values on the stack.

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

stack over time

a

a b

a b c

a (b c /)

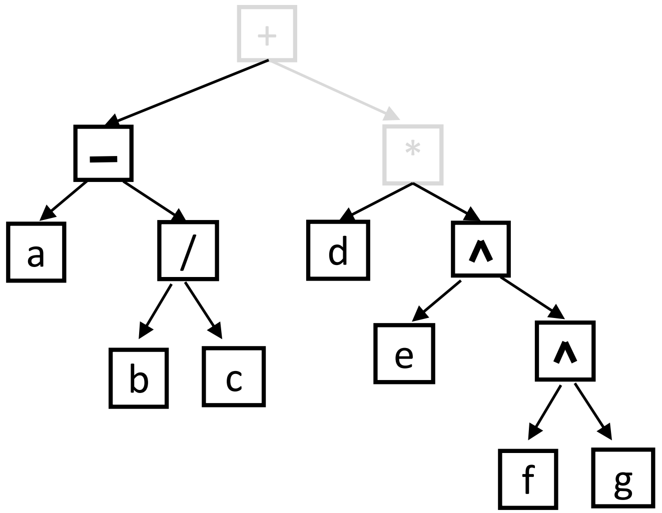
(a (b c /) -)

:

(a (b c /) -) d e f g

(a (b c /) -) d e (f g ^)

(a (b c /) -) d (e (f g ^) ^)



Three values on the stack.

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

a

a b

a b c

a (b c /)

(a (b c /) -)

:

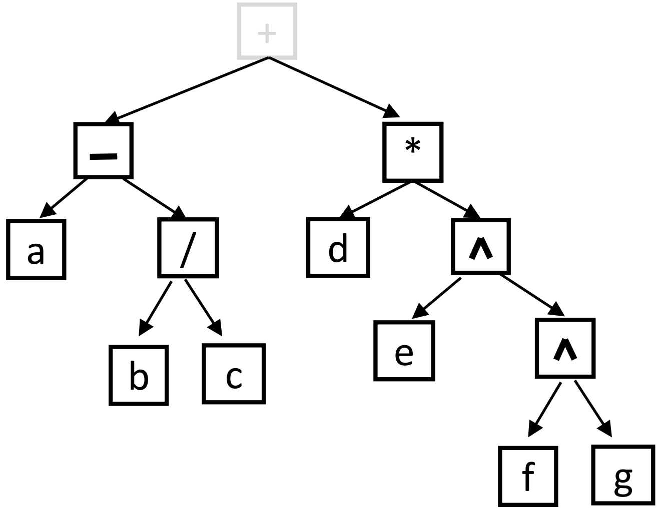
(a (b c /) -) d e f g

(a (b c /) -) d e (f g ^)

(a (b c /) -) d (e (f g ^) ^)

(a (b c /) -) (d (e (f g ^) ^) *)

stack over time

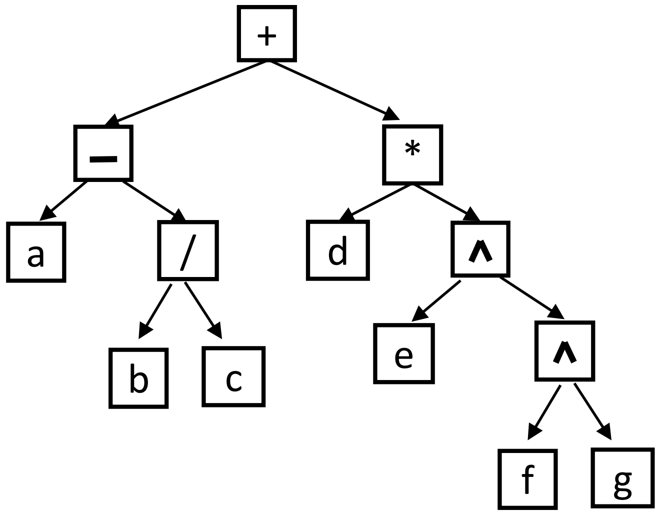


Two values on the stack.

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

stack
over
time

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



One value on the stack (the result)

Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of "tokens".

```
s = empty stack
cur =  first token of expression list
while (cur != null){
    if ( cur  is a base expression )
        s.push( cur )
    else{                                  //  cur is an operator



    }
    cur = cur.next
}
```

Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of "tokens".

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