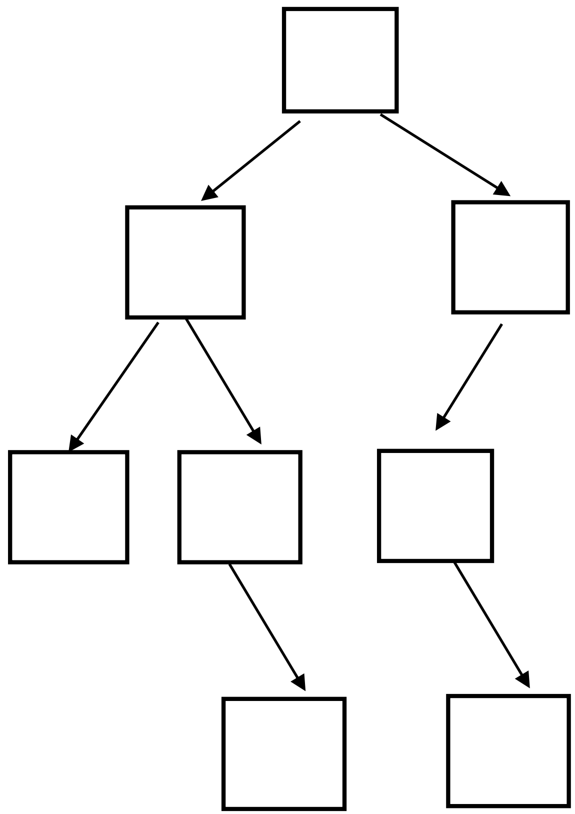
COMP 250

Lecture 21

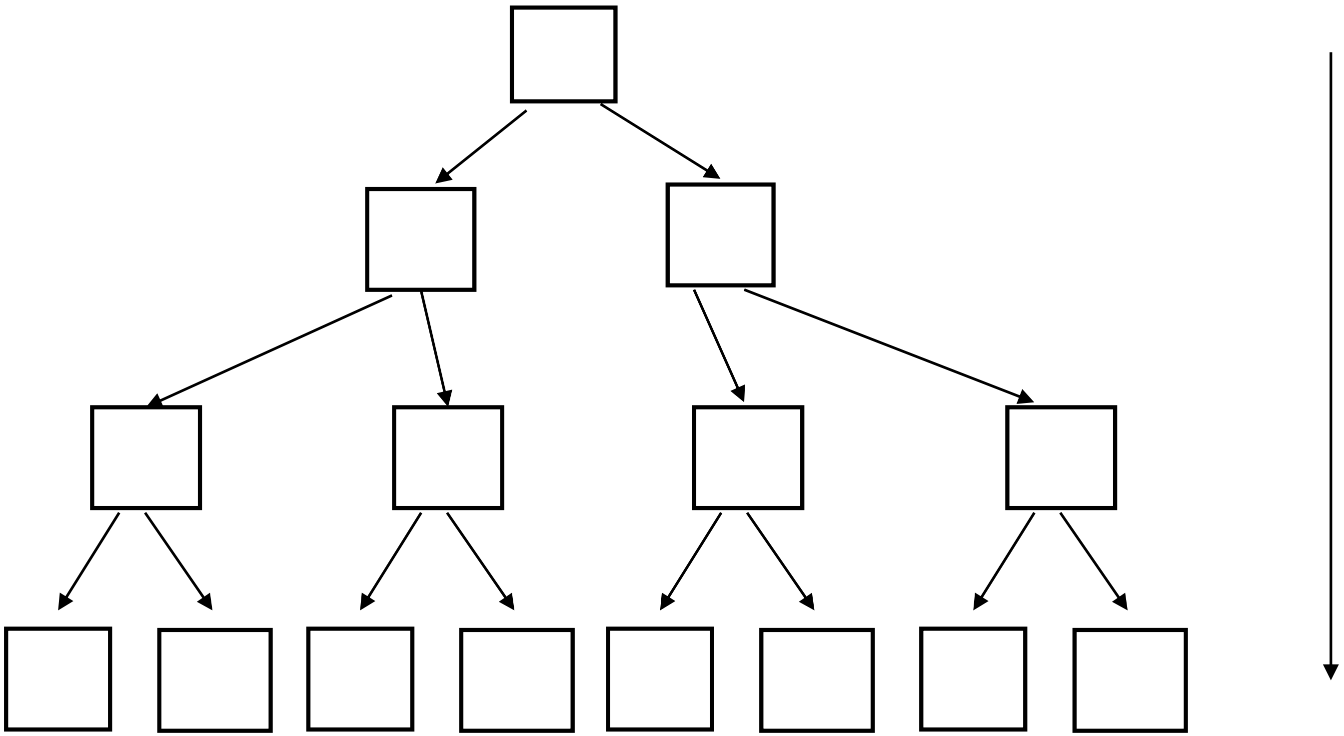
binary trees, expression trees

Oct. 27, 2017

Binary tree:
each node has at most two children.

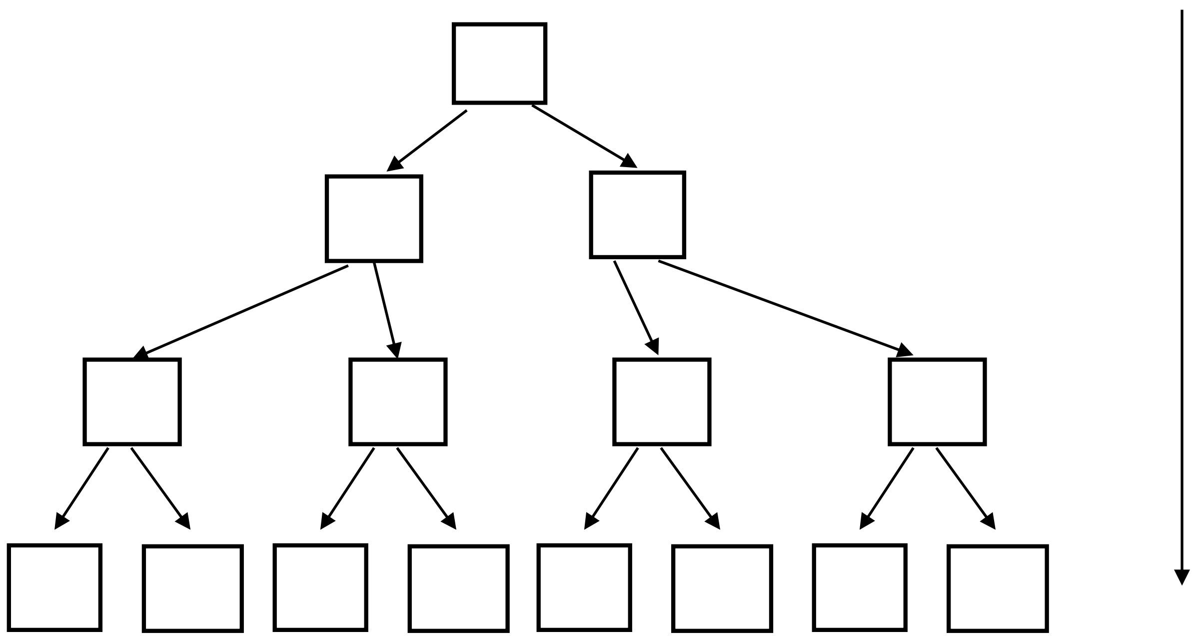


Maximum number of nodes in a binary tree?



Height h
(e.g. 3)

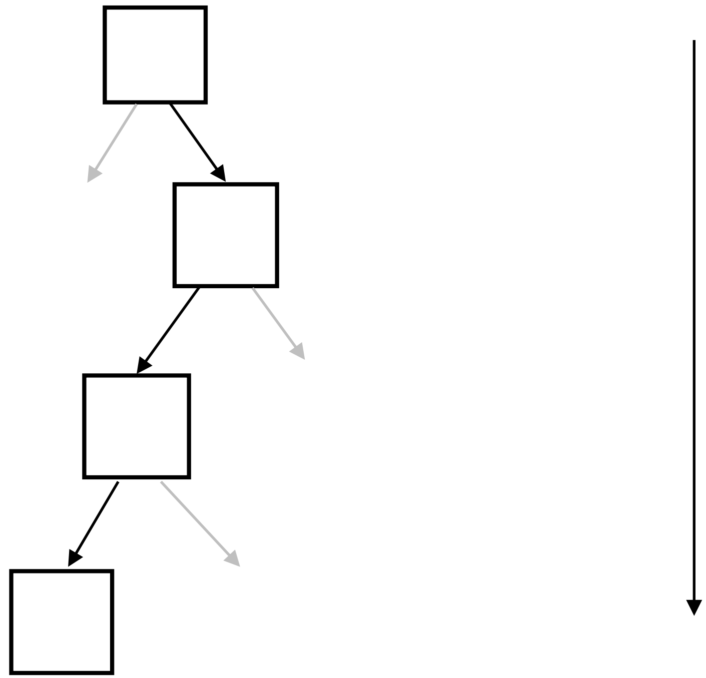
Maximum number of nodes in a binary tree?



Height h
(e.g. 3)

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

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

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

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){

}
```

```
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
    }
}
```

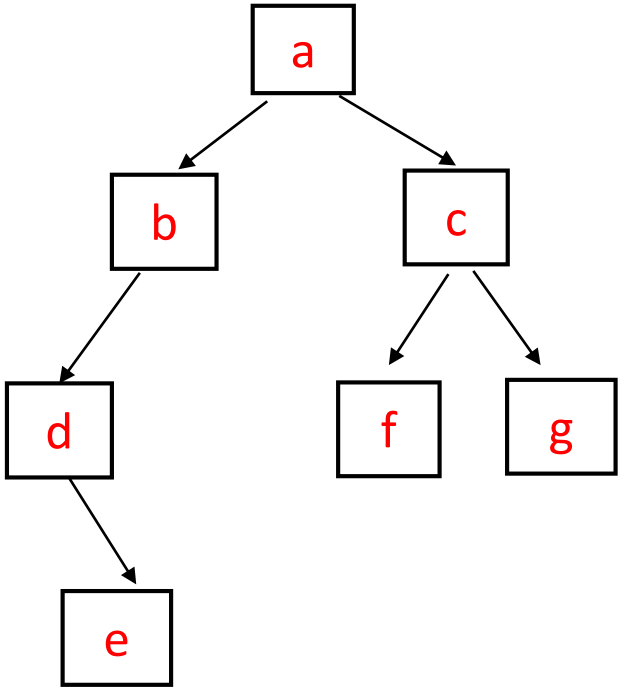
```
inorderBT (root){

}
```

```
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

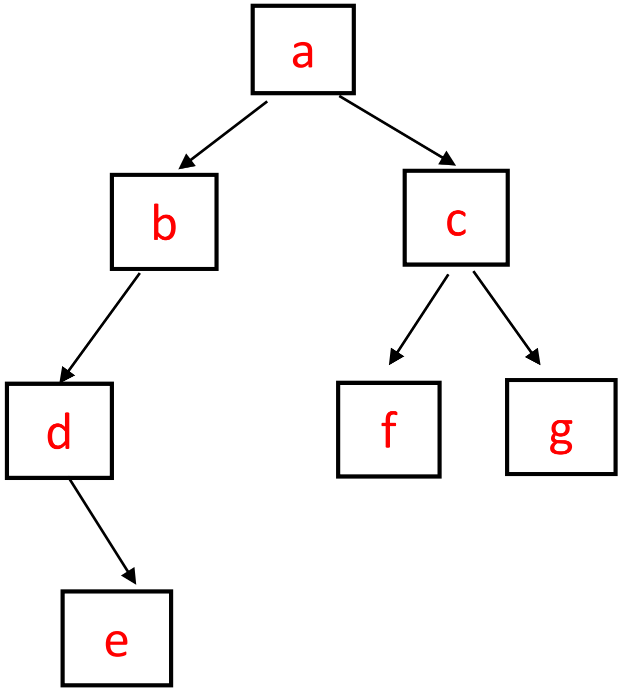


Pre order:

In order:

Post order:

Example

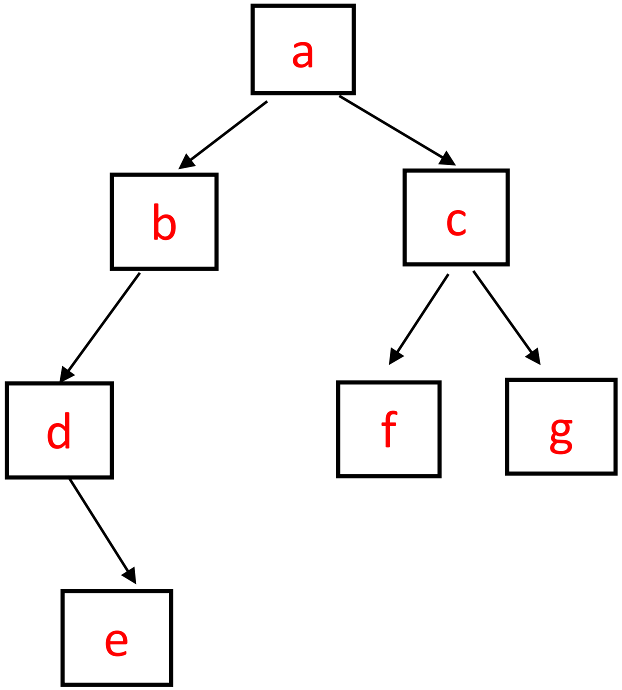


Pre order: a b d e c f g

In order:

Post order:

Example

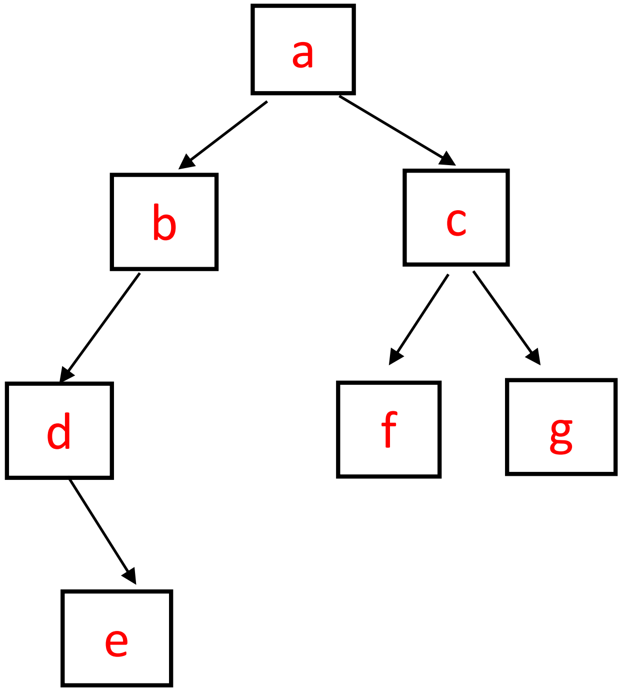


Pre order: a b d e c f g

In order: d e b a f c g

Post order:

Example



Pre order: a b d e c f g

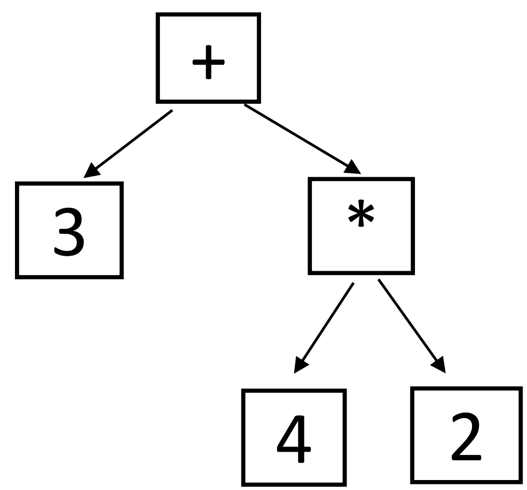
In order: d e b a f c g

Post order: e d b f g c a

Expression Tree

e.g. 3 + 4 * 2

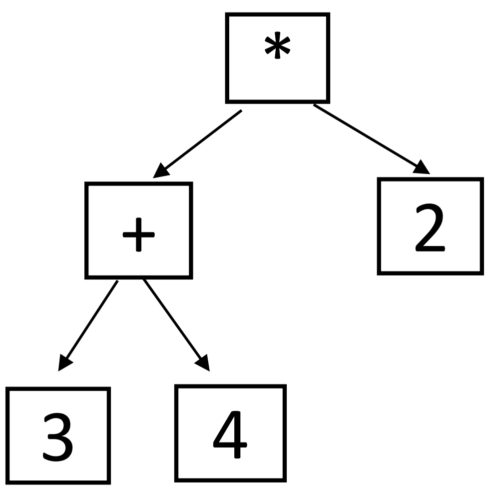
3 + (4 * 2)



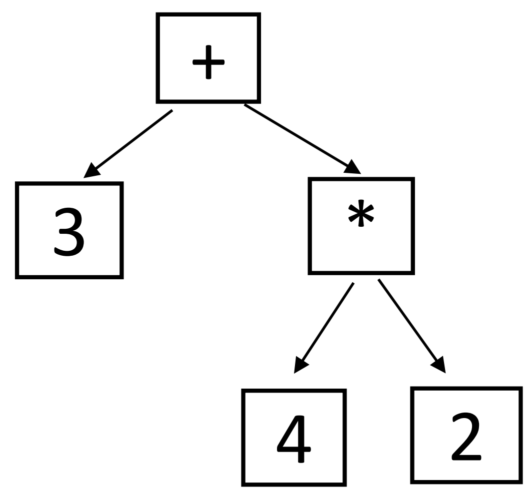
Expression Tree

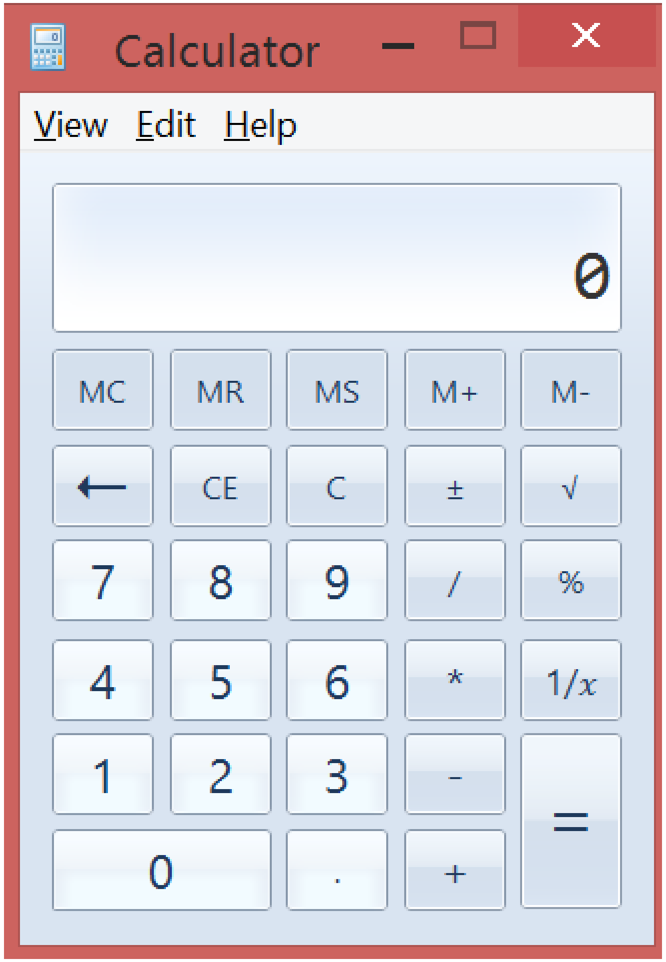
e.g. 3 + 4 * 2

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

^ is exponentiation: $e \wedge f \wedge g = e \wedge (f \wedge g)$

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

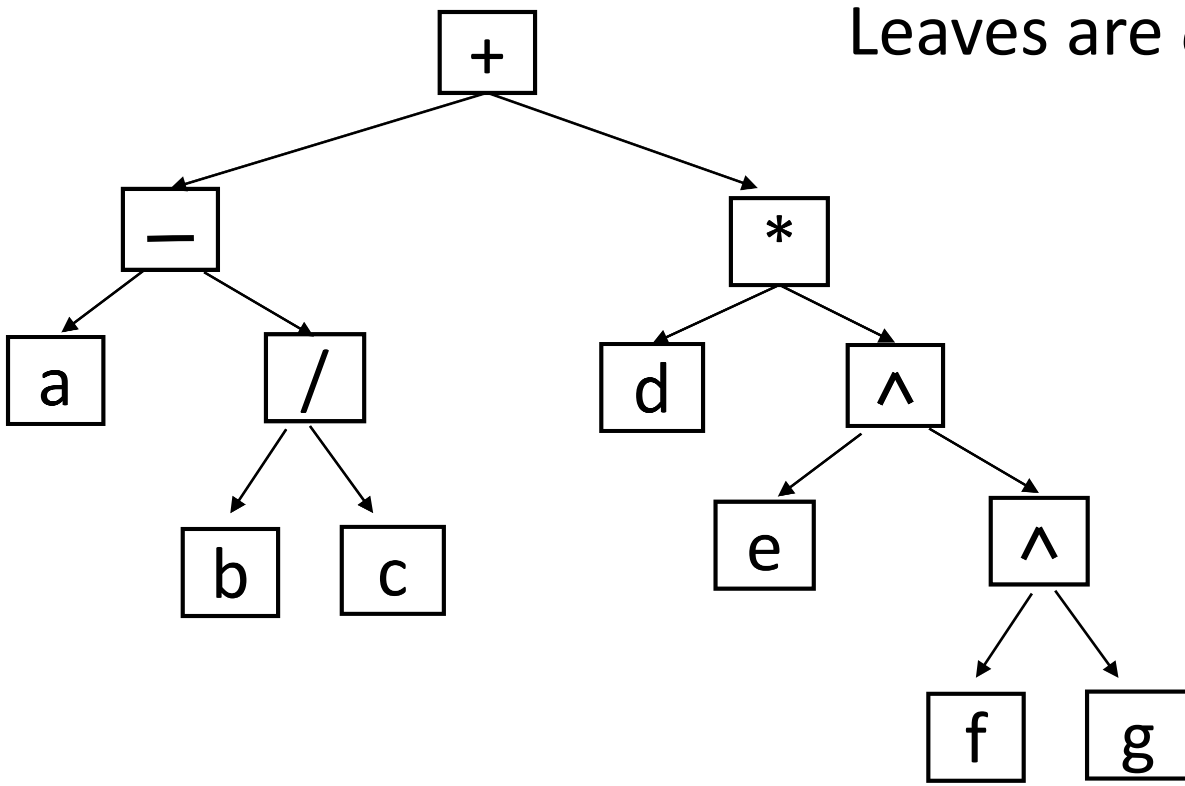
Operator precedence ordering makes brackets unnecessary.

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

Expression Tree

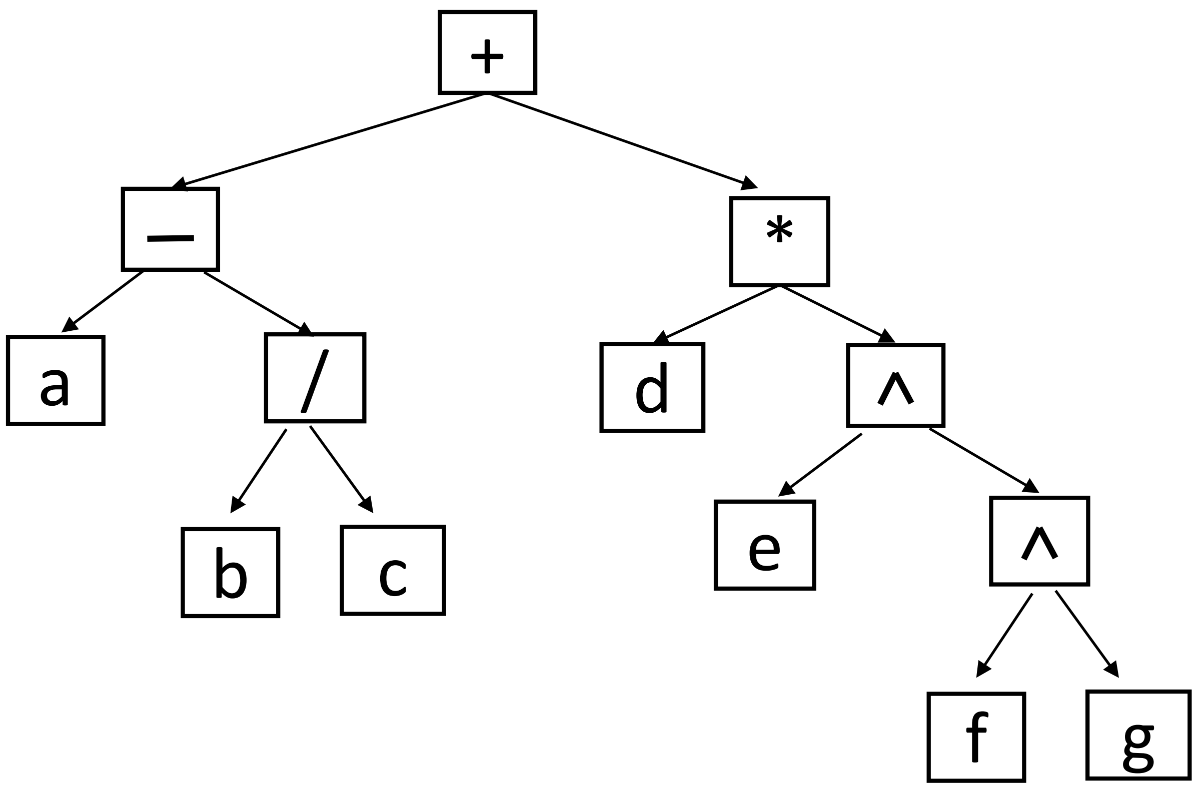
$$a - b \,/\, c + d * e \wedge f \wedge g \;\equiv\; (a - (b \,/\, c)) + (d * (e \wedge (f \wedge g)))$$

Internal nodes are *operators*.
Leaves are *operands*.

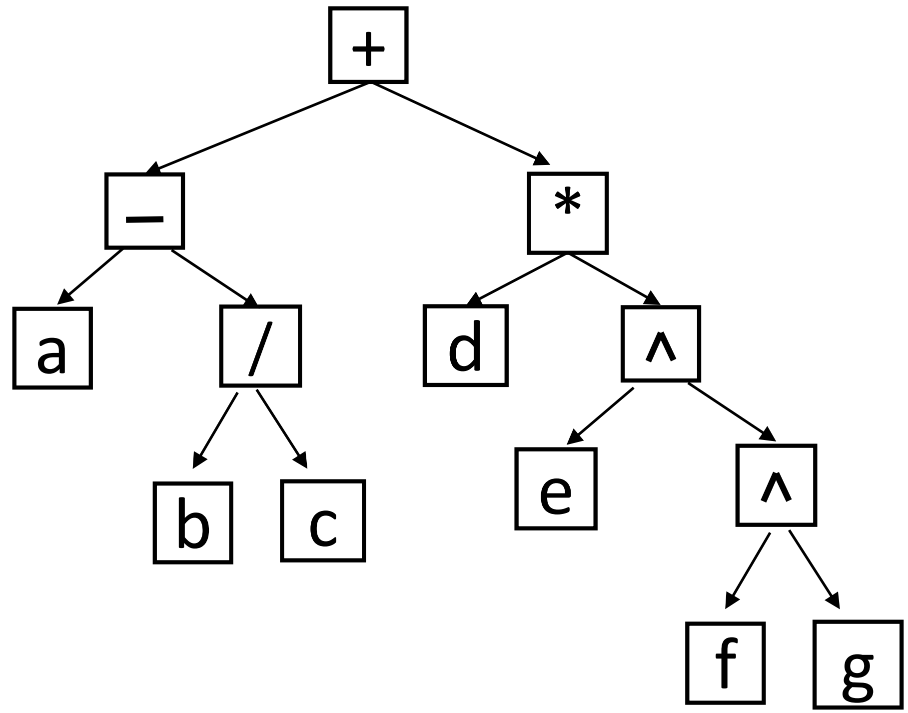


An expression tree can be a way of *thinking about* the ordering of operations used when evaluating an expression.

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

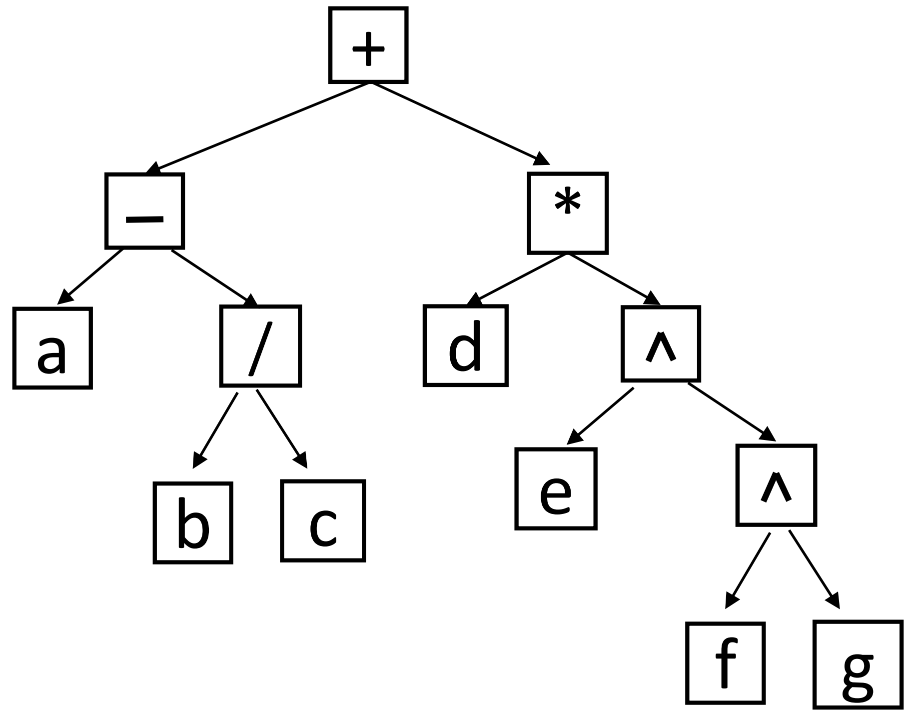


If we traverse an expression tree, and *print out* the node label, what is the expression printed out?



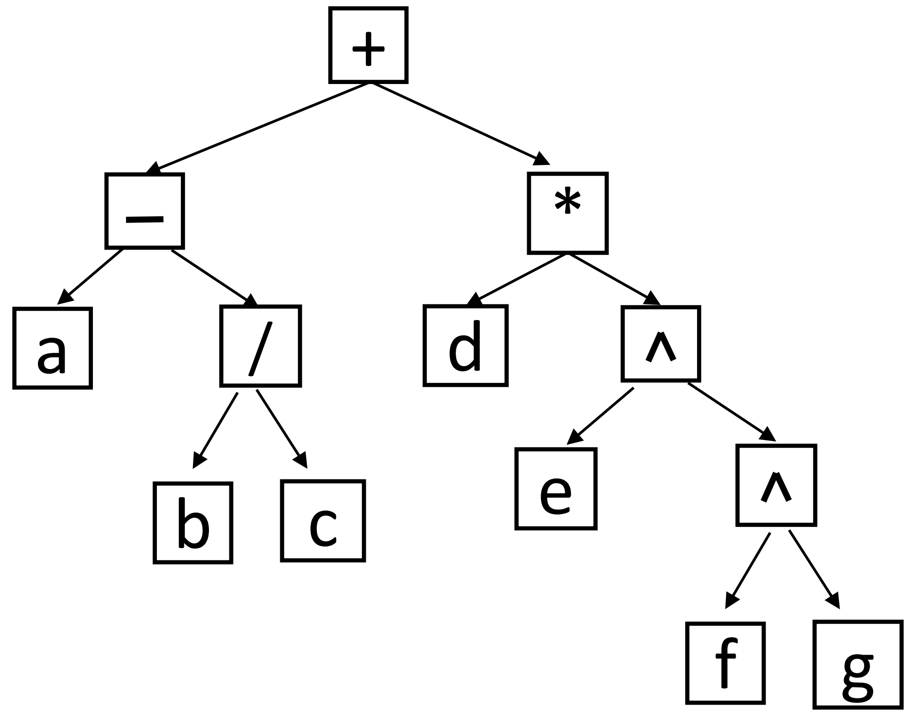
preorder traversal gives

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

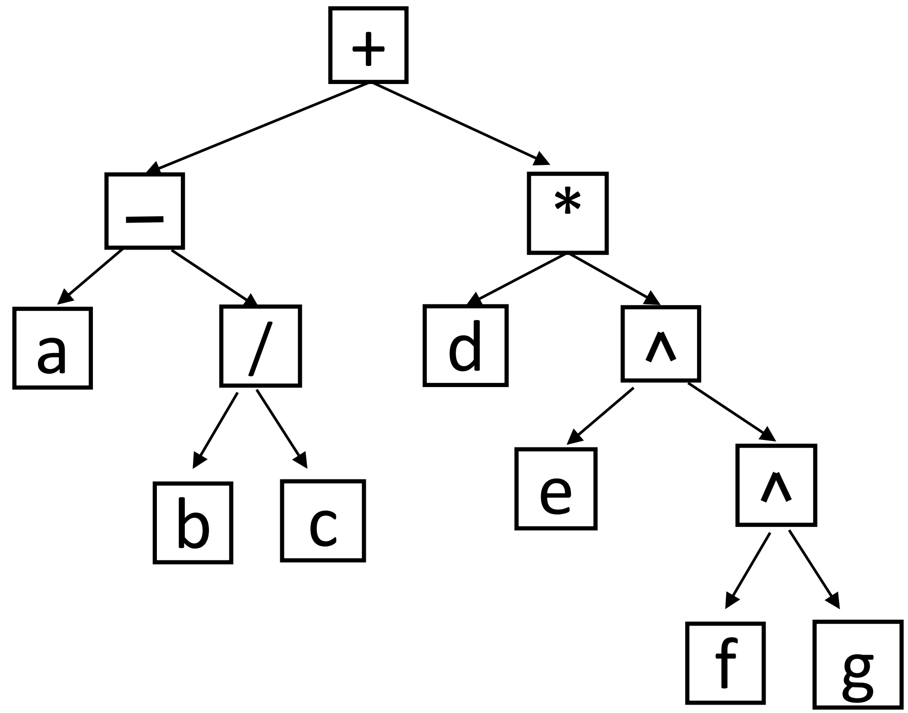
If we traverse an expression tree, and print out the node label, what is the expression printed out?



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

inorder traversal gives :

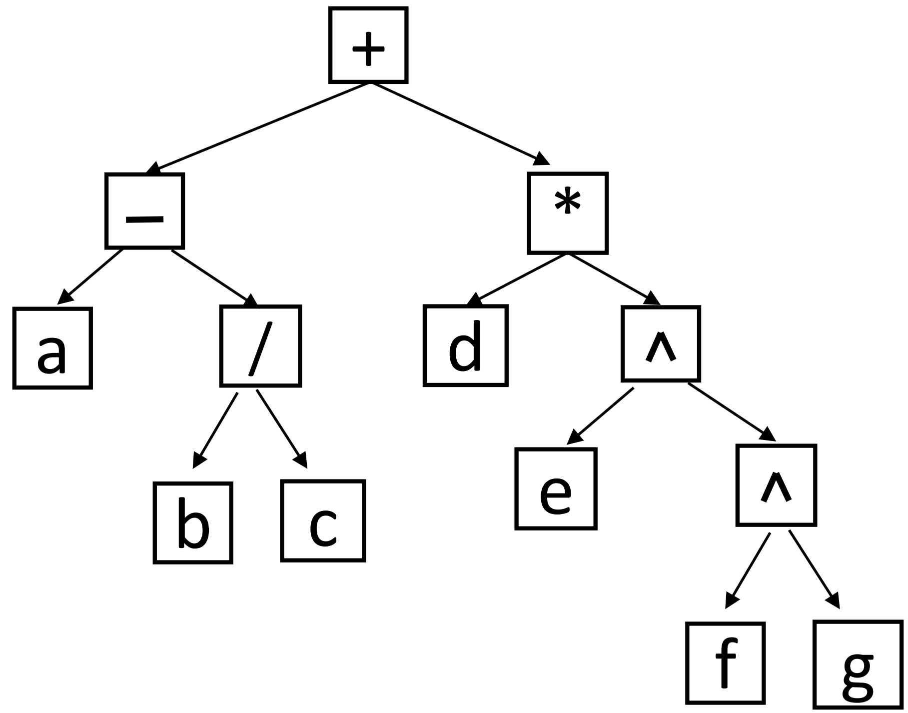
If we traverse an expression tree, and print out the node label, what is the expression printed out?



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?

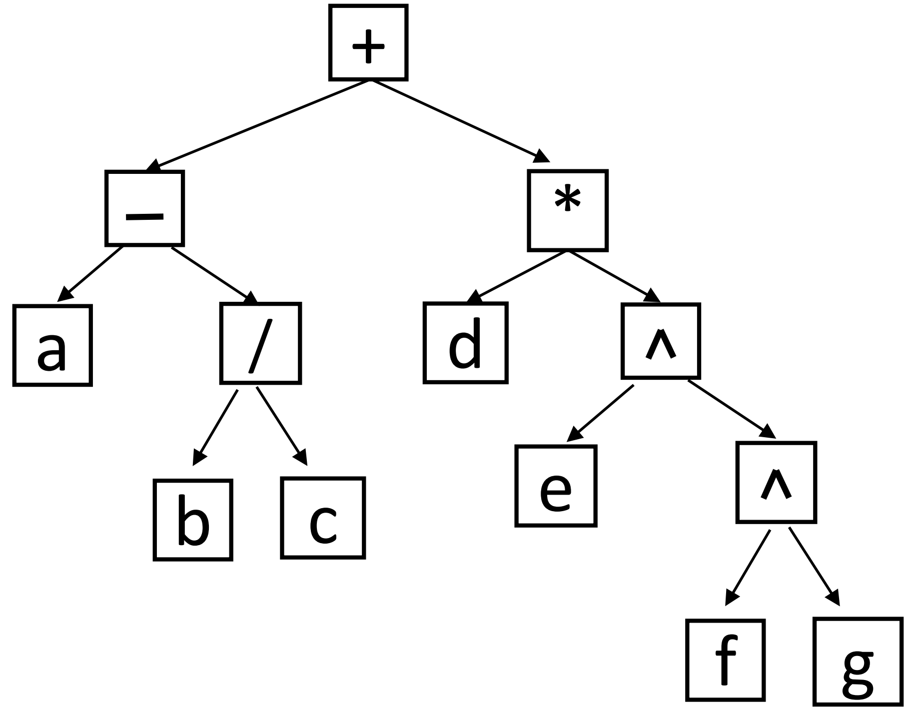


preorder traversal gives : + – a / b c * d ^ e ^ f g

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?

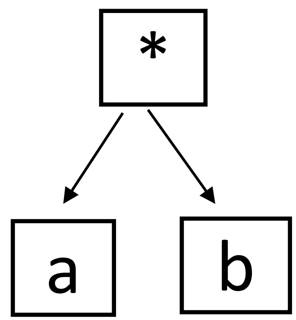


preorder traversal gives : + – a / b c * d ^ e ^ f g

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 *

Infix, prefix, postfix expressions

baseExp = variable | integer

op = + | - | * | / | ^

preExp = baseExp | op preExp preExp

Infix, prefix, postfix expressions

baseExp = variable | integer

op = + | - | * | / | ^

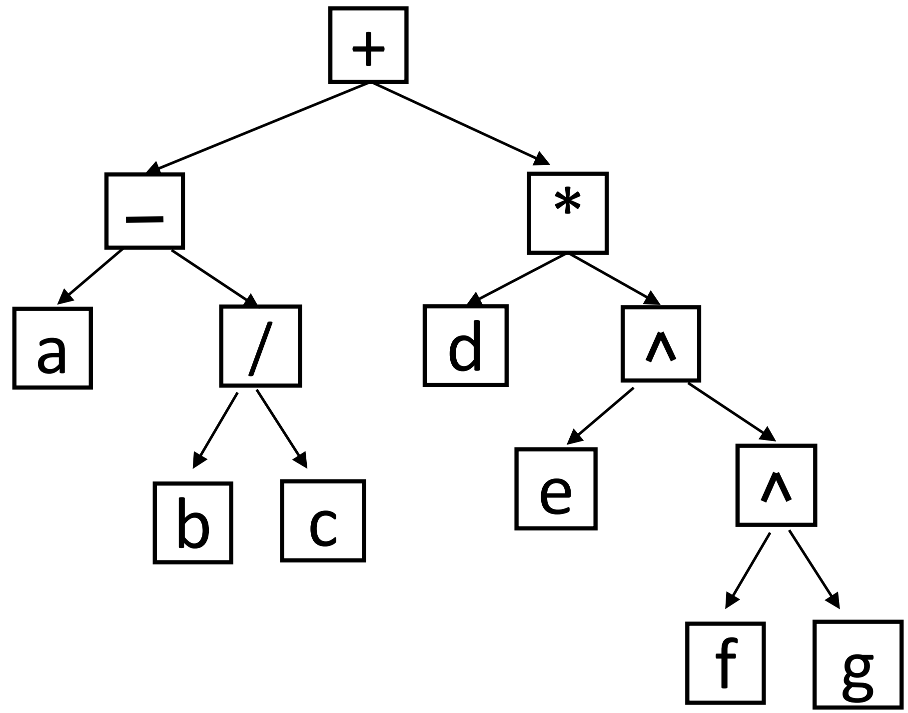
preExp = baseExp | op preExp prefExp

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)



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)

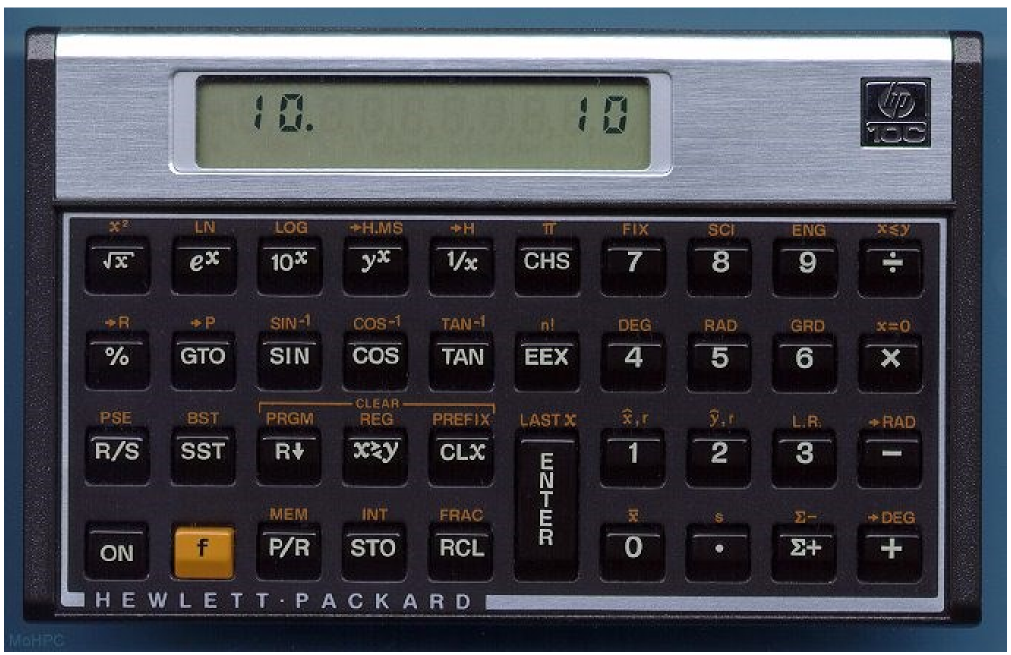
Some calculators (esp. Hewlett Packard) require users to input expressions using RPN.

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>

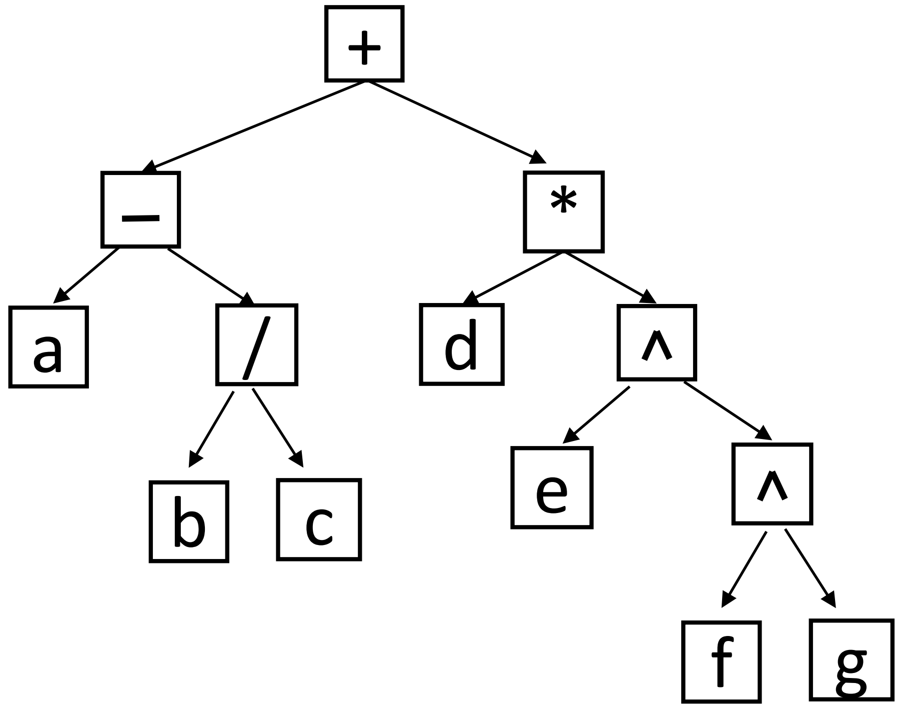
* <enter>

3 <enter>

+ <enter>

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{   //  the root is an operator
      firstOperand       =  evalExpTree( root.leftchild )
      secondOperand   =  evalExpTree( root.rightchild )
      return evaluate(firstOperand, root, secondOperand)
    }
}
```

What if we are not given an expression tree?

Infix expressions are awkward to evaluate because of precedence ordering.

Infix expressions with brackets are relatively easy to evaluate   e.g. Assignment 2.

Assignment 2 (ignore case of ++, --)

```
for each token in expression {
     if token is a number
          valueStack.push(token)
     else if token is ")" {    //   then you have a binary expression

     }
}
return valueStack.pop()
```

Assignment 2 (ignore case of ++, --)

```
for each token in expression {
     if token is a number
            valueStack.push(token)
     else if token is ")" {    //  then you have a binary expression
             operator   =  opStack.pop()
             operand2  =  valueStack.pop()
             operand1  =  valueStack.pop()
             numStack.push(  operand1 operator operand2)
      }
}
return valueStack.pop()
```

Infix expressions *with brackets* are relatively easy to evaluate e.g. with two stacks as in Assignment 2.

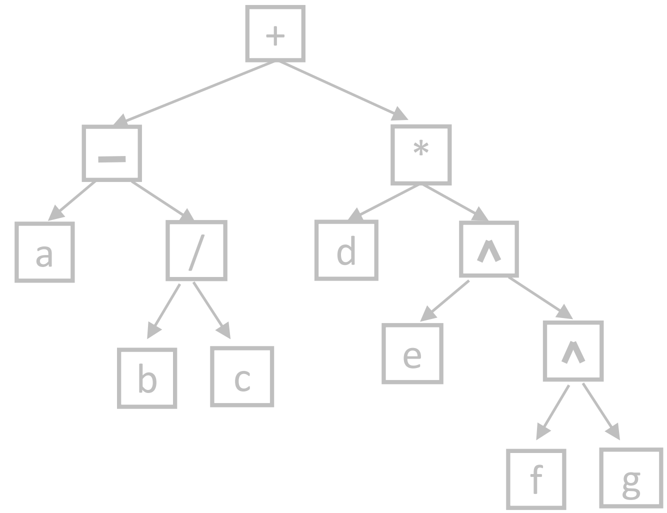
Postfix expressions *without brackets* are easy to evaluate. Use one stack, namely for values (not operators).

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:

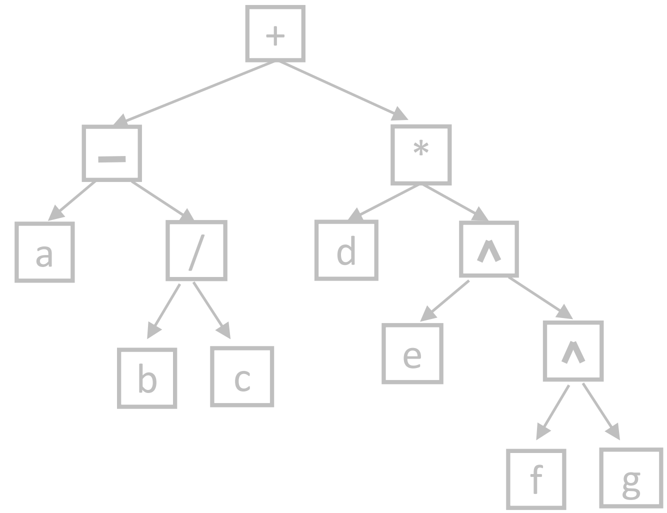
a b c / - d e f g ^ ^ * +

stack over time

a

a b

a b c



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 ^ ^ * +

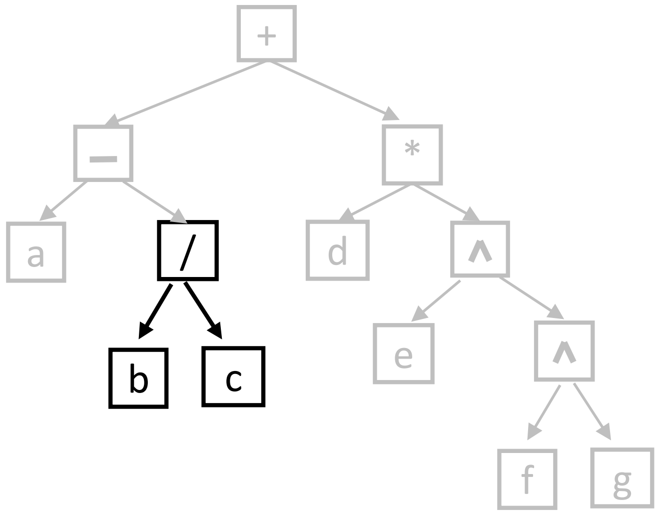
stack over time

a

a b

a b c

a (b c /)

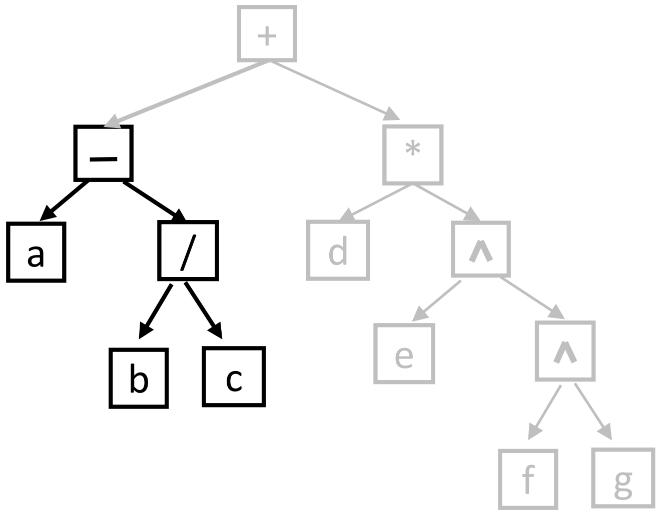


We don’t push operator onto stack.
Instead we pop value twice, evaluate, and push.

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

stack
over
time

a
a b
a b c
a (b c /)
(a (b c /) -)



Now there is one value on the stack.

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

stack over time

a

a b

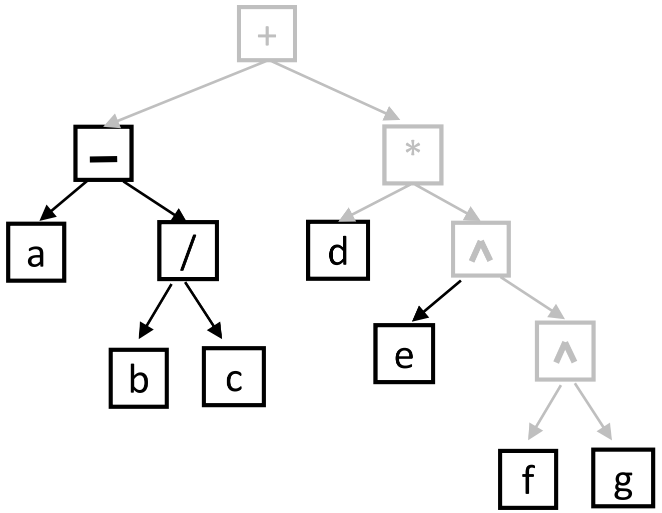
a b c

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 /)

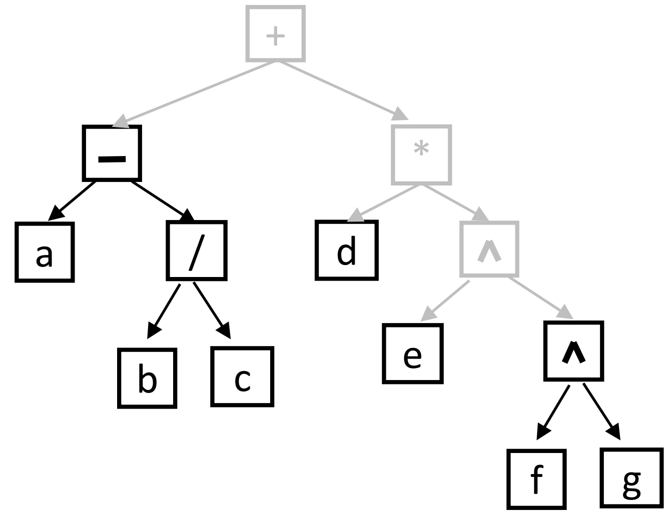
(a (b c /) -)

:

(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 /)

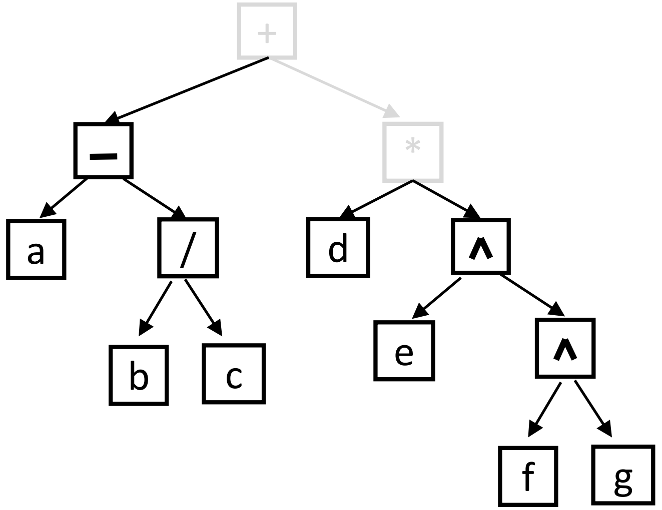
(a (b c /) -)

:

(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 ^ ^ * +

stack over time

a

a b

a b c

a (b c /)

(a (b c /) -)

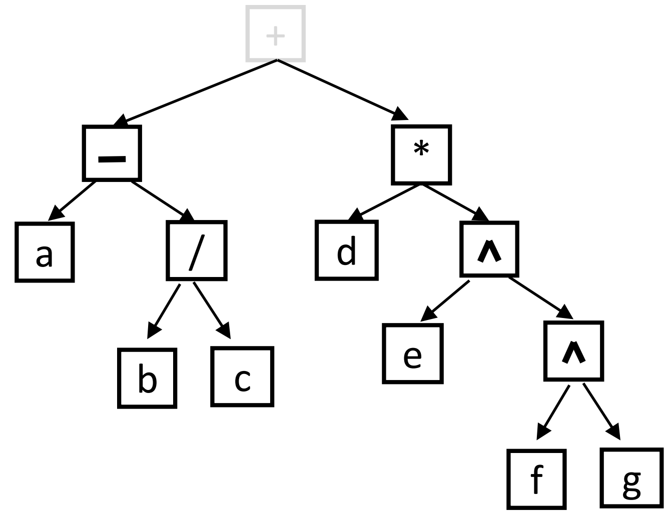
:

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

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

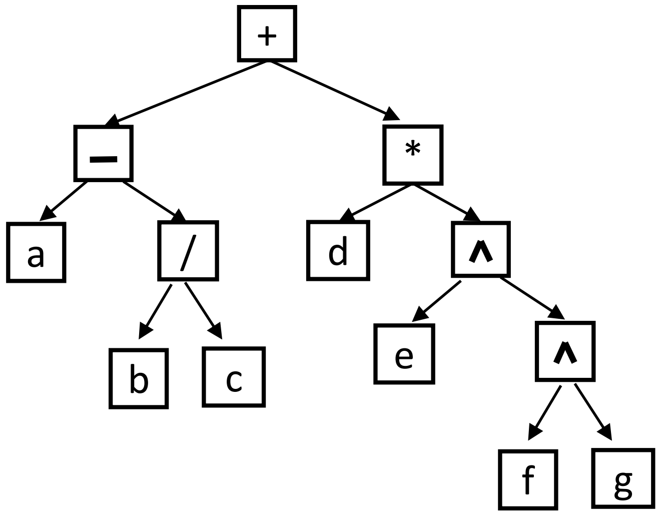
(a (b c /) -) d (e (f g ^) ^)

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



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.

Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of elements.

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



    }
    cur = cur.next
}
```

Algorithm: Use a stack to evaluate a postfix expression

Let expression be a list of elements.

```
s = empty stack
cur =  first element 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
}
```

ASIDE

There are many variations of expression tree problems.

e.g. Define an algorithm that computes a postfix expression *directly* from an infix expression *with no brackets.* This is not so obvious if you have to respect precedence ordering e.g. +, -, *, /, ^

http://wcipeg.com/wiki/Shunting_yard_algorithm