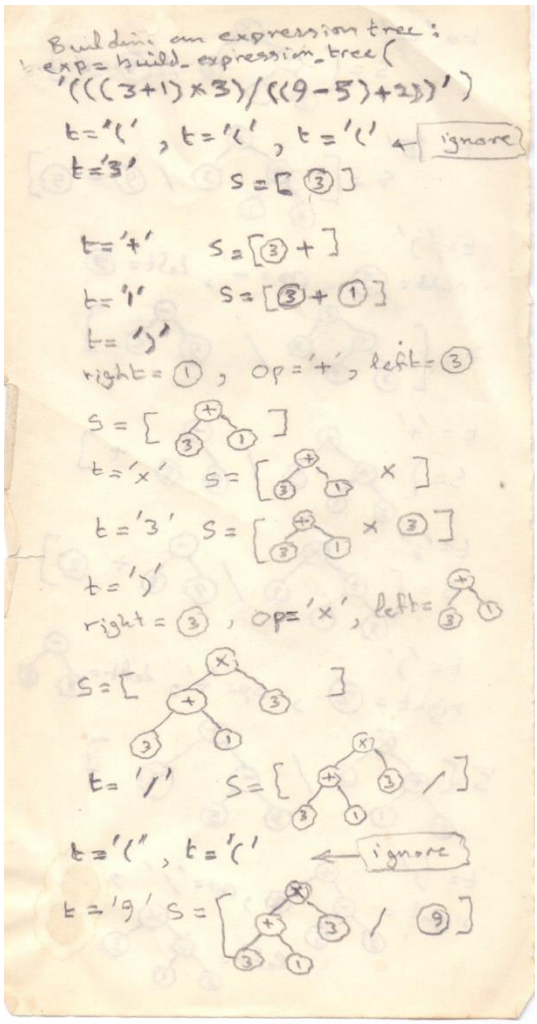
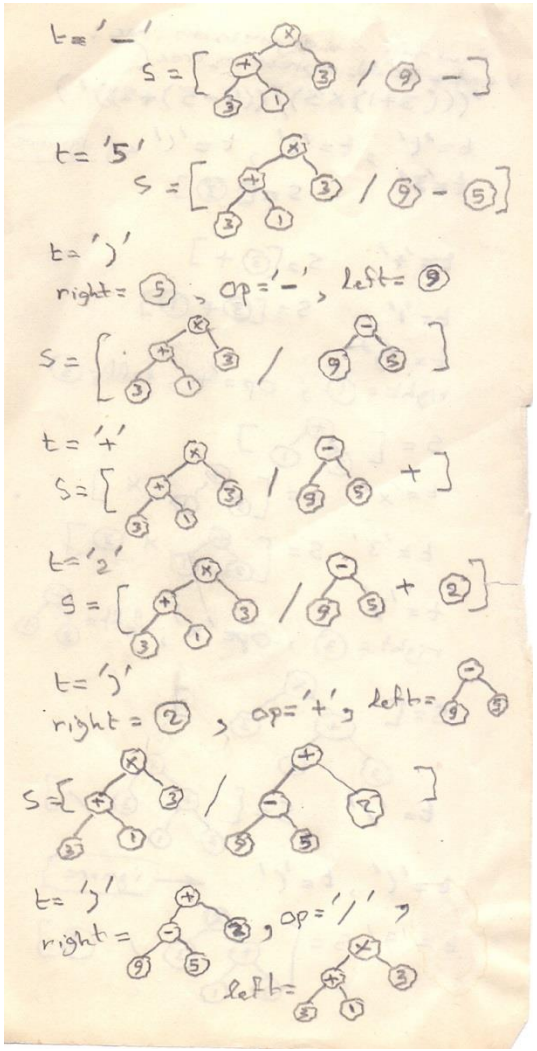
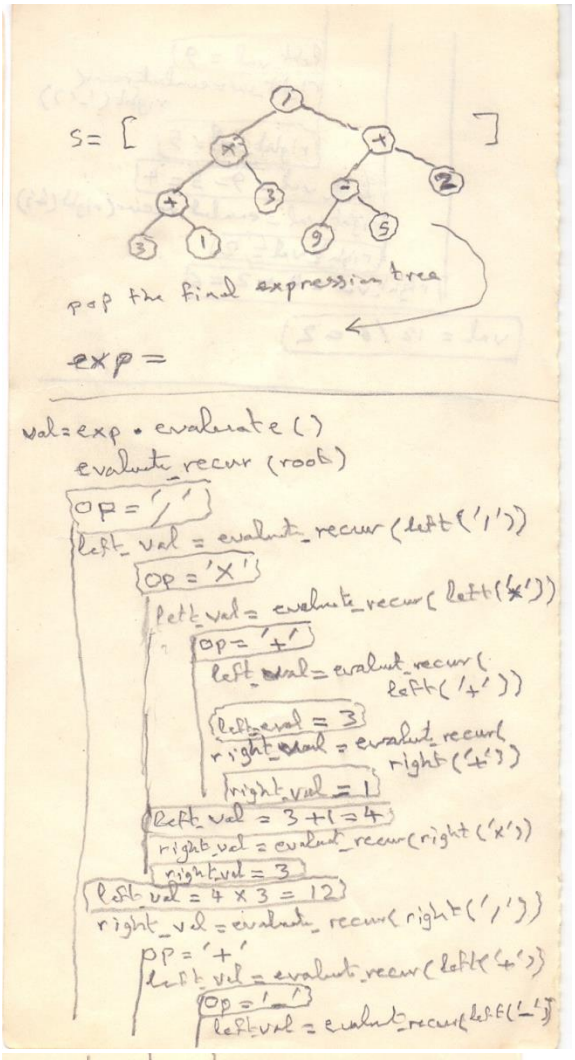
```
from LinkedBinaryTree import LinkedBinaryTree
class ExpressionTree(LinkedBinaryTree):
 """An arithmetic expression tree."""
 def __init__(self, token, left=None, right=None):
  """Create an expression tree.
  In a single parameter form, token should be a leaf value (e.g.,'42'),
  and the expression tree will have that value at an isolated node.
  In a three-parameter version, token should be an operator,
  and left and right should be existing ExpressionTree instances
  that become the operands for the binary operator.
  """
  super().__init__( ) # LinkedBinaryTree initialization
  if not isinstance(token, str):
   raise TypeError('Token must be a string')
  self._add_root(token) # use inherited, nonpublic method
  if left is not None: # presumably three-parameter form
   if token not in '+-*x/':
    raise ValueError('token must be valid operator')
   self._attach(self.root( ), left, right) # use inherited, nonpublic method

 def __str__(self):
  """Return string representation of the expression."""
  pieces = [ ] # sequence of piecewise strings to compose
  self._parenthesize_recur(self.root( ), pieces)
  return ''.join(pieces)

 def _parenthesize_recur(self, p, result):
  """Append piecewise representation of p's subtree to resulting list."""
  if self.is_leaf(p):
   result.append(str(p.element( ))) # leaf value as a string
  else:
   result.append('(') # opening parenthesis
   self._parenthesize_recur(self.left(p), result)   # left subtree
   result.append(p.element()) # operator
   self._parenthesize_recur(self.right(p), result) # right subtree
   result.append(')') # closing parenthesis

 def evaluate(self):
  """Return the numeric result of the expression."""
  return self._evaluate_recur(self.root( ))

 def _evaluate_recur(self, p):
  """Return the numeric result of subtree rooted at p."""
  if self.is_leaf(p):
   return float(p.element( )) # we assume element is numeric
  else:
   op = p.element( )
   left_val = self._evaluate_recur(self.left(p))
```

```
  right_val = self._evaluate_recur(self.right(p))
  if op == '+':
   return left_val + right_val
  elif op == '-':
   return left_val - right_val
  elif op == '/':
   return left_val / right_val
  else:
   return left_val * right_val # treat 'x' or '*' as multiplication


def build_expression_tree(tokens):
 """Returns an ExpressionTree based upon a tokenized expression."""
 S = [ ] # we use Python list as stack
 for t in tokens:
  if t in '+-x*/' : # t is an operator symbol
   S.append(t) # push the operator symbol
  elif t not in '()' : # consider t to be a literal
   S.append(ExpressionTree(t)) # push trivial tree storing value
  elif t == ')' : # compose a new tree from three constituent parts
   right = S.pop( ) # right subtree as per LIFO
   op = S.pop( ) # operator symbol
   left = S.pop( ) # left subtree
   S.append(ExpressionTree(op, left, right)) # repush tree
   # we ignore a left parenthesis
 return S.pop( )


S = '(((30+20)x40)/((90-50)+10))'
print(tokenize(S))
exp = build_expression_tree(tokenize(S))
print(exp)
print(exp.evaluate())
```

Build Expression Tree Example




S= [ ]
/
x
+
+
3
-
2
+
1
9
5
3
pop the final expression tree
exp =
val = exp.evaluate()
evaluate_recur(root)
op = '/'
left_val = evaluate_recur(left('/'))
op = 'x'
left_val = evaluate_recur(left('x'))
op = '+'
left_val = evaluate_recur(left('+'))
left_val = 3
right_val = evaluate_recur(right('+'))
right_val = 1
left_val = 3+1 = 4
right_val = evaluate_recur(right('x'))
right_val = 3
left_val = 4 x 3 = 12
right_val = evaluate_recur(right('/'))
op = '+'
left_val = evaluate_recur(left('+'))
op = '-'
left_val = evaluate_recur(left('-'))
left_val = 9
right_val = evaluate_recur(right('-'))
right_val = 5
left_val = 9-5 = 4
right_val = evaluate_recur(right('+'))
right_val = 2
right_val = 4+2 = 6
val = 12/6 = 2