CSCI 204: Data Structures & Algorithms

Revised by Xiannong Meng based on textbook author's notes

Binary Tree Application Expression Tree

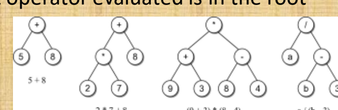
Revised based on textbook author's notes.

Expression Trees

- A binary tree in which the operators are stored in the interior nodes and the operands are sored in the leaves.
  - Used to evaluate an expression.
  - Used to convert an infix expression to either prefix or postfix notation.
- We've learned to evaluate expressions using stacks. The tree implementation will be a good contrast to that of a stack.

Expression Trees

- The tree structure is based on the order in which the operators are evaluated.
  - Operators in lower-level nodes are evaluated first.
  - The last operator evaluated is in the root node.



Expression Tree ADT

- An expression tree is a binary tree representation of an arithmetic expression.
  - Contains various operators (+, -, *, /, %)
  - Contains operands comprised of single integer digits and single-letter variables.

- ExpressionTree(exp_str)
- evaluate(var_dict)
- *__str__()*

Expression Tree Example

- We can use the ADT to evaluate basic arithmetic expressions of any size.

```
# Create a dictionary containing values for the variables.
vars = { 'a' : 5, 'b' : 12 }

# Build the tree for a sample expression and evaluate it.
exp_tree = ExpressionTree( "(a/(b-3))" )
print( "The result = ", exp_tree.evaluate(vars) )

# We can change the value assigned to a variable
# and reevaluate.
vars['a'] = 22
print( "The result = ", exp_tree.evaluate(vars) )
```

Try ex1.py

Expression Tree Implementation

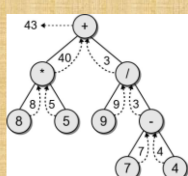
exptree.py

```
class ExpressionTree :
  def __init__( self, exp_str ):
    self._exp_tree = None
    self._build_tree( exp_str )   # recursion

  def evaluate( self, var_map ):
    return self._eval_tree( self._exp_tree, var_map ) # recursion

  def __str__( self ):
    return self._build_string( self._exp_tree )
# ...

# Storage class for creating the tree nodes.
class _ExpTreeNode :
  def __init__( self, data ):
    self.element = data
    self.left = None
    self.right = None
```


Expression Tree Evaluation

- We can develop an algorithm to evaluate the expression.
  - Each subtree represents a valid subexpression.
  - Lower-level subtrees have higher precedence.
  - For each node, the two subtrees must be evaluated first.
- How does it work?

Evaluation Call Tree



Expression Tree Implementation

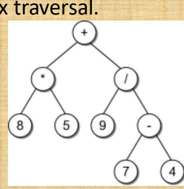
exptree.py

```
class ExpressionTree :
# ...
  def _eval_tree( self, subtree, var_dict ):
     # See if the node is a leaf node
    if subtree.left is None and subtree.right is None :
      # Is the operand a literal digit?
      if subtree.element >= '0' and subtree.element <= '9' :
        return int(subtree.element)
      else :   # Or is it a variable?
        assert subtree.element in var_dict, "Invalid variable."
        return var_dict[subtree.element]

     # Otherwise, it's an operator that needs to be computed.
    else :   # post-order traversal!
      # Evaluate the expression in the subtrees.
      lvalue = _eval_tree( subtree.left, var_dict )
      rvalue = _eval_tree( subtree.right, var_dict )

      # Evaluate the operator using a helper method.
      return compute_op( lvalue, subtree.element, rvalue )
```


String Representation

- To convert an expression tree to a string, we must perform an infix traversal.

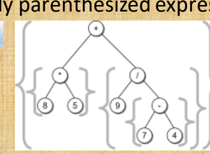
```
8 * 5 + 9 / 7 - 4
```



String Representation

- The result was not correct because required parentheses were missing.
  - Can easily create a fully parenthesized expression.

```
((8 * 5) + (9 / (7 - 4)))
```



Expression Tree Implementation

exptree.py

```python
class ExpressionTree :
# ...
  def _build_string( self, tree_node ):
     # If the node is a leaf, it's an operand.
    if tree_node.left is None and tree_node.right is None :
      return str( tree_node.element )

     # Otherwise, it's an operator.
    else :   # in-order traversal!
      exp_str = '('
      exp_str += self._build_string( tree_node.left )
      exp_str += str( tree_node.element )
      exp_str += self._build_string( tree_node.right )
      exp_str += ')'
      return exp_str
```

13

Expression Tree Construction

- An expression tree is constructed by parsing the fully-parenthesized expression and examining the tokens.
  - New nodes are inserted as the tokens are examined.
  - Each set of parentheses will consist of:
    - an interior node for the operator
    - two children either single valued or a subexperssion.

14

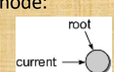
Expression Tree Construction

- For simplicity, we assume:
  - the expression is stored in a string with no white space.
  - the expression is valid and fully parenthesized.
  - each operand will be a single-digit or single-letter variable.
  - the operators will consist of +, -, *, /, %

15

Expression Tree Construction

- Consider the expression `(8*5)`
- The process starts with an empty root node set as the current node:

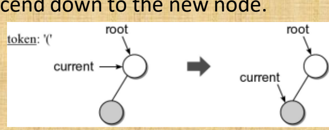


- The action at each step depends on the current token.

16

Expression Tree Construction

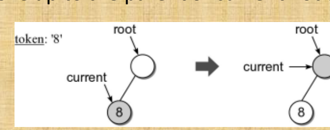
- When a left parenthesis is encountered: `(8*5)`
  - a new node is created and linked as the left child of the current node.
  - descend down to the new node.



17

Expression Tree Construction

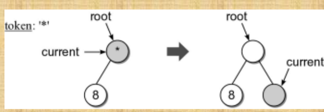
- When an operand is encountered: `(8*5)`
  - the data field of the current node is set to contain the operand.
  - move up to the parent of current node.



18

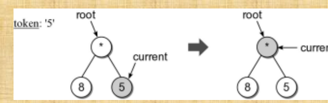
Expression Tree Construction

- When an operator is encountered: (8*5)
  - the data field of the current node is set to the operator.
  - a new node is created and linked as the right child of the current node.
  - descend down to the new node.



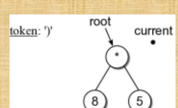
Expression Tree Construction

- Another operand is encountered: (8*5)



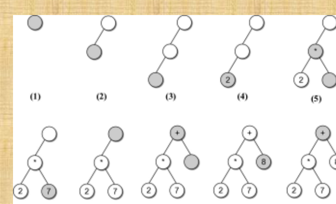
Expression Tree Construction

- When a right parenthesis: (8*5)
  - move up to the parent of the current node.



Expression Example #2

- Consider another expression: ((2*7)+8)



Expression Tree Implementation

exptree.py

```
class ExpressionTree :
# ...
  def _build_tree( self, exp_str ):
     # Build a queue containing the tokens from the expression.
    expQ = Queue()
    for token in exp_str :
      expQ.enqueue( token )

     # Create an empty root node.
    self._exp_tree = _ExpTreeNode( None )

     # Call the recursive function to build the tree.
    self._rec_build_tree( self._exp_tree, expQ )
```


Expression Tree Implementation

exptree.py

```
class ExpressionTree :
# ...
  def _rec_build_tree( self, cur_node, expQ ):
     # Extract the next token from the queue.
    token = expQ.dequeue()
     # See if the token is a left paren: '('
    if token == '(' :
      cur_node.left = _ExpTreeNode( None )
      build_treeRec( cur_node.left, expQ )
       # The next token will be an operator: + - / * %
      cur_node.data = expQ.dequeue()
      cur_node.right = _ExpTreeNode( None )
      self._build_tree_rec( cur_node.right, expQ )
       # The next token will be a ), remove it.
      expQ.dequeue()

     # Otherwise, the token is a digit.
    else :
      cur_node.element = token
```