Expression Trees
Expression Trees

- A binary tree in which the operators are stored in the interior nodes and the operands are stored in the leaves.
  - Used to evaluate an expression.
  - Used to convert an infix expression to either prefix or postfix notation.
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 (+, -, *, /, %) (only consider binary operators).
  - Assumes operands comprised of single integer digits and single-letter variables.
  - Within a fully parenthesized expression.

| ExpressionTree( expStr ) |
| evaluate( varDict ) |
| toString() |
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.
expTree = expressionTree( "(a/(b-3))" )
print( "The result = ", expTree.evaluate(vars) )

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

```python
class ExpressionTree:
    def __init__(self, expStr):
        self._expTree = None
        self._buildTree(expStr)

    def evaluate(self, varMap):
        return self._evalTree(self._expTree, varMap)

    def __str__(self):
        return self._buildString(self._expTree)

# ... 

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

exptree.py
String Representation

- To convert an expression tree to a string, we must perform an inorder traversal.
String Representation

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

\[
((8 \times 5) + (9 \div (7 - 4)))
\]
Expression Tree Implementation

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

    # Otherwise, it's an operator.
    else:
      expStr = '('
      expStr += self._buildString(treeNode.left)
      expStr += str(treeNode.element)
      expStr += self._buildString(treeNode.right)
      expStr += ')'
      return expStr
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.
class ExpressionTree:
    # ...
    def _evalTree(self, subtree, varDict):
        # 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 varDict, "Invalid variable."
                return varDict[subtree.element]
        else:
            # Otherwise, it's an operator that needs to be computed.
            lvalue = _evalTree(subtree.left, varDict)
            rvalue = _evalTree(subtree.right, varDict)

            # Evaluate the operator using a helper method.
            return computeOp(lvalue, subtree.element, rvalue)
Evaluation Call Tree
Expression Tree Construction

- An expression tree is constructed by parsing the 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 subexpression.
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 +, -, *, /, %
Expression Tree Construction

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

  ![Diagram of an empty root node](image)

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

- When a left parenthesis is encountered: \((8 \times 5)\)
  - a new node is created and linked as the left child of the current node.
  - descend down to the new node.
Expression Tree Construction

- When an operand is encountered: \((8 \times 5)\)
  - the data field of the current node is set to contain the operand.
  - move up to the parent of current node.
Expression Tree Construction

- When an operator is encountered: \((8 \times 5)\)
  - 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.

Diagram:
- Initial node with value 8
- Operator node '*'
- New node created as the right child of the operator node
- Descend down to the new node

Diagram:
- Initial node with value 8
- Operator node '*'
- New node created as the right child of the operator node
Expression Tree Construction

- Another operand is encountered: \((8 \times 5)\)
Expression Tree Construction

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

```
<table>
<thead>
<tr>
<th>token: ')'</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
</tr>
<tr>
<td>current</td>
</tr>
</tbody>
</table>

```

- Diagram:

```
[Diagram of an expression tree with nodes labeled 8, *, 5, and a right parenthesis symbol.]
```
Expression Example #2

- Consider another expression: 

\[(2 \times 7) + 8\]
class ExpressionTree:
    # ...
    def _buildTree(self, expStr):
        # Build a queue containing the tokens from the expression.
        expQ = Queue()
        for token in expStr:
            expQ.enqueue(token)

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

        # Call the recursive function to build the tree.
        self._recBuildTree(self._expTree, expQ)
class ExpressionTree:
    # ...
    def _recBuildTree( self, curNode, expQ ):
        # Extract the next token from the queue.
        token = expQ.dequeue()
        # See if the token is a left paren: '('
        if token == '(
            curNode.left = _ExpTreeNode( None )
            self._recBuildTree( curNode.left, expQ )
            # The next token will be an operator: + - / * %
            curNode.data = expQ.dequeue()
            curNode.right = _ExpTreeNode( None )
            self._recBuildTree( curNode.right, expQ )
            # The next token will be a ), remove it.
            expQ.dequeue()

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