```
printf \\n\\n\\n            #print three blank lines



cat p6.sh                  #display the shell script file for the program
#!/bin/bash

set -v                       #turn on echo
printf \\n\\n\\n            #print three blank lines
cat p6.sh                  #display the shell script file for the program
printf \\f                    #issue a form feed (top of a new page)
cat -b p6.java           #display the source file with line numbers
:                             #null command
:
:
javac p6.java           #compile the java file
java p6                    #execute the file from the current directory
:
:
:
date                         #print the date
printf \\f                    #issue a form feed (top of a new page)

cat -b p6.java           #display the source file with line numbers
     1    /*
     2    ---------------------------------------------------------------------
     3    PROGRAM NAME: Program 6
     4    PROGRAMMER:   Samuel Jentsch
     5    CLASS:        CSC 241.001, Fall 2013
     6    INSTRUCTOR:   Dr. D. Dunn
     7    DATE STARTED: November 10, 2013
     8    DUE DATE:     November 11, 2013
     9    REFERENCES:   Computer Science
    10                      Data Abstraction and Problem Solving with Java
    11                      Janet J. Prichard & Frank M. Carrano
    12                  Dr. Dunn: assignment information sheet

    13    PROGRAM PURPOSE:
    14    a. This program reads a series of infix expressions.
    15    b. The infix expressions are processed into postfix expressions
    16         using stacks, evaluated, stored in a binary tree, and
    17         displayed in preorder using a preorder traversal of the
    18         tree.
```

```
19    ADTs:
20       Stack

21    FILES USED:
22       p6a.dat - a file containing infix expressions.
23       p6b.dat - a file containing variable values.
24
25    ---------------------------------------------------------------------
26     */

27    import java.io.*;
28    import java.util.*;

29    public class p6 {
30
31         public static void main(String[] args) {
32              File infixFile = new File("../instr/p6a.dat");
33              File variableFile = new File("../instr/p6b.dat");

34              handleFileInput(infixFile, variableFile);
35
36         }

37         public static void handleFileInput(File infixFile, File varFile) {
38              //-------------------------------------------------------------------------------------------
39              //Reads infix expressions as lines from the file parameter passed. Call processInfixExpression to
40              //manipulate the infix expressions as they  are read. Handles the expressions as necessary to
41              //evaluate, convert, etc.
42              //Preconditions: Two files passed as parameters. infixFile contains lines of infix expressions,
43              //                   varFile contains variables followed by their value separated by lines.
44              //Postconditions: As the file is processed, methods in the program are called to:
45              //                  -convert the expression into a postfix expression.
46              //                  -create a tree from the postfix expression and traverse the tree in preorder.
47              //                  -Evaluate the postfix expression and display the value.
48              //-------------------------------------------------------------------------------------------
49
50              ArrayList<String> varValues = new ArrayList<String>();
51
52              try {
53                   Scanner fileReader = new Scanner(varFile);
54                   while(fileReader.hasNext())
55                        varValues.add(fileReader.next());
```

```
56              } catch (FileNotFoundException e) {
57                   // TODO Auto-generated catch block
58                   e.printStackTrace();
59              }
60
61              try {
62                   Scanner fileReader = new Scanner(infixFile);
63                   while(fileReader.hasNext()) {
64                        String infix = fileReader.nextLine();
65                        System.out.println(infix);
66                        String pFix = processInfixExpressionIntoPostfix(infix);
67                        System.out.println(pFix);
68                        TreeNode<String> treeRoot = createTreeFromPostfix(pFix);
69                        preorderTraversal(treeRoot);
70                        System.out.println();
71                        System.out.printf("Result: %.4f\n", evaluatePostfixWithVariableValues(pFix, varValues));
72                        System.out.println();
73                   }
74              } catch (FileNotFoundException e) {
75                   // TODO Auto-generated catch block
76                   e.printStackTrace();
77              }
78         }
79
80         public static String processInfixExpressionIntoPostfix(String infixExpression) {
81              //-------------------------------------------------------------------------------------------
82              //Process the infixExpression passed. Use isOperator and
83              //getPrecedence to build a postfix expression from the
84              //infix expression using a stack. Processes the expression
85              //character by character and adds to the postfix expression
86              //string to build a postfix from the infix.
87              //Preconditions: String passed as parameter containing an infix expression.
88              //Postcondition: A string is returned representing the postfix version
89              //                        of the infix expression passed.
90              //-------------------------------------------------------------------------------------------
91
92              Stack<String> s = new Stack<String>();
93              String postfixExpression = "";
94              for(int i = 0; i < infixExpression.length(); i++) {
95                   String op = infixExpression.charAt(i) + "";
96
97                   if(!isOperator(op)) {
98                        postfixExpression += op;
```

```
 99                  } else {
100                       int precedence = getPrecedence(op);
101
102                       if(op.matches("[)]")) {
103                            while(!s.isEmpty() && !s.peek().matches("[(]")) {
104                                 postfixExpression += s.pop();
105                            }
106                            s.pop();
107                       } else{
108                            while(!s.isEmpty() && precedence < getPrecedence(s.peek())
109                                      && !s.peek().matches("[(]")) {
110                                 postfixExpression += s.pop();
111                            }
112                            s.push(op);
113                       }
114                  }
115             }
116
117             while(!s.isEmpty())
118                  postfixExpression += s.pop();
119
120             return postfixExpression;
121        }
122
123        public static boolean isOperator(String op) {
124             //-------------------------------------------------------------------------------------------
125             //Takes a string as a parameter and returns
126             //true if it is one of 7 operator values.
127             //returns false otherwise.
128             //Preconditions: String passed a parameter.
129             //Postconditions: Returns true if op is a recognized operator,
130             //                       False if not.
131             //-------------------------------------------------------------------------------------------
132
133             boolean isOperator = false;
134
135             if(op.matches("[*, /, ^, (, ), +, -]"))
136                  isOperator = true;
137
138             return isOperator;
139        }
140
141        public static int getPrecedence(String op) {
```

```
142             //-------------------------------------------------------------------------------------------
143             //Return the integer precedence value for the
144             //operator passed as a parameter.
145             //Preconditions: String op passed as parameter.
146             //Postconditions: Returns the integer precedence value for the passed operator.
147             //-------------------------------------------------------------------------------------------
148
149             int precedence = 1;
150
151             if(op.matches("[(, )]"))
152                  precedence = 5;
153             else if(op.matches("^"))
154                  precedence = 4;
155             else if(op.matches("[*, /]"))
156                  precedence = 3;
157
158             return precedence;
159         }
160
161         public static double evaluatePostfixWithVariableValues(String postfixExpression, ArrayList<String>
irValues) {
162             //-------------------------------------------------------------------------------------------
163             //Process postfix expressions passed as parameter. Use stack to evaluate the expression and the
164             //varValues ArrayList passed to the method to acquire values for variable in expression. Use
165             //performOperationsWithOperands() to get the value of the operation being processed with the
166             //appropriate operands. Return the value of the evaluated expression.
167             //
168             //If an operand is encountered, push to stack. If operator is encountered, pop two operands off of
169             //stack, look up variable value if necessary, and evaluate the operands with the operator. Push
170             //the new value onto the stack.
171             //Preconditions: postfixExpression passed as String, ArrayList passed containing variables
172             //followed by their values.
173             //Postconditions: The value of evaluating the expression is returned.
174             //-------------------------------------------------------------------------------------------
175             double value = 0;
176
177             if(postfixExpression.length() == 1) {
178                  String v1 = postfixExpression.charAt(0) + "";
179                  try {
180                       value = Double.parseDouble(v1);
181                  } catch(Exception ex) {
182                       int varIndex = varValues.indexOf(v1);
183                       value = Double.parseDouble(varValues.get(varIndex + 1));
```

```
184                  }
185                  return value;
186             }
187
188             Stack<String> s = new Stack<String>();
189             for(int i = 0; i < postfixExpression.length(); i++) {
190                  String op = postfixExpression.charAt(i) + "";
191                  if(isOperator(op)) {
192                       String v2 = s.pop();
193                       String v1 = s.pop();
194
195                       double v1Double = 0.0;
196                       try {
197                            v1Double = Double.parseDouble(v1);
198                       } catch(Exception ex) {
199                            int varIndex = varValues.indexOf(v1);
200                            v1Double = Double.parseDouble(varValues.get(varIndex + 1));
201                       }
202
203                       double v2Double = 0.0;
204                       try {
205                            v2Double = Double.parseDouble(v2);
206                       } catch(Exception ex) {
207                            int varIndex = varValues.indexOf(v2);
208                            v2Double = Double.parseDouble(varValues.get(varIndex + 1));
209                       }
210
211                       s.push(performOperationWithOperands(op, v1Double, v2Double) + "");
212                  } else {
213                       s.push(op);
214                  }
215             }
216
217             if(!s.isEmpty())
218                  value = Double.parseDouble(s.pop());
219
220             return value;
221        }
222
223        public static double performOperationWithOperands(String op, double v1, double v2) {
224             //-------------------------------------------------------------------------------------------
225             //This method uses a switch statement based on the
226             //op parameter passed. The correct operation is
```

```
227             //applied to the double parameters and the value
228             //after the operation is returned.
229             //Preconditions: String operator, two double values passed.
230             //Postconditions: Returns the value of performing the passed operator with the passed operands.
231             //-----------------------------------------------------------------------------------------
232
233             double value = 0.0;
234
235             switch (op.charAt(0)) {
236             case '+':
237                  value = v1 + v2;
238                  break;
239             case '-':
240                  value = v1 - v2;
241                  break;
242             case '^':
243                  value = Math.pow(v1, v2);
244                  break;
245             case '*':
246                  value = v1 * v2;
247                  break;
248             case '/':
249                  value = v1 / v2;
250                  break;
251             default:
252                  System.out.println("Unsupported operation: " + op);
253             }
254
255             return value;
256         }
257
258         public static TreeNode<String> createTreeFromPostfix(String postfix) {
259             //-----------------------------------------------------------------------------------------
260             //Process a postfix expression passed as a parameter and
261             //turns it into a binary algebraic expression tree using
262             //a stack. Creates treeNodes using operators and operands.
263             //Preconditions: postfix expression passed as string.
264             //Postconditions: Processes postfix expression and creates tree using a stack. Algorithm is same
265             //as that for postfix evaluation. Returns the root node of the created tree.
266             //-----------------------------------------------------------------------------------------
267
268             Stack<TreeNode<String>> s = new Stack<TreeNode<String>>();
269             for(int i = 0; i < postfix.length(); i++) {
```

```
270                  String op = postfix.charAt(i) + "";
271                  if(isOperator(op)) {
272                       TreeNode<String> v2 = s.pop();
273                       TreeNode<String> v1 = s.pop();
274
275                       TreeNode<String> newNode = new TreeNode<String>(op, v1, v2);
276                       s.push(newNode);
277                  } else {
278                       s.push(new TreeNode<String>(op));
279                  }
280             }
281
282             return s.pop();
283        }
284
285        public static void preorderTraversal(TreeNode<String> treeNode) {
286             //-----------------------------------------------------------------------------------------------
287             //Traverse the tree referenced by the root node
288             //passed as a parameter recursively in preorder.
289             //Preconditions: root to a tree passed as parameter.
290             //Postconditions: Tree referenced by treeNode is traversed in preorder and printed.
291             //-----------------------------------------------------------------------------------------------
292
293             if(treeNode != null) {
294                  System.out.print(treeNode.item);
295                  preorderTraversal(treeNode.leftChild);
296                  preorderTraversal(treeNode.rightChild);
297             }
298        }
299
300   }

301   class TreeNode<T> {
302        /*
303        ------------------------------------------------------------------
304        CLASS NAME: Tree Node
305        VARIABLE BANK:
306             item - T, data item
307             leftChild - TreeNode, reference to the left child.
308             rightChild - TreeNode, reference to the right child.
309        DESCRIPTION:
310             This class is the basis for a binary tree node. It allows for
311             a left and right child and a single data item
```

```
312         ---------------------------------------------------------------------
313          */
314
315         T item;
316         TreeNode<T> leftChild;
317         TreeNode<T> rightChild;
318
319         public TreeNode(T newItem) {
320              //Create a TreeNode with the passed data item.
321              this.item = newItem;
322              this.leftChild = null;
323              this.rightChild = null;
324         }
325
326         public TreeNode(T newItem, TreeNode<T> left, TreeNode<T> right) {
327              this.item = newItem;
328              this.leftChild = left;
329              this.rightChild = right;
330         }
331
332    }//end TreeNode

333    class Stack<T> {
334         /*
335         ---------------------------------------------------------------------
336         CLASS NAME: Stack
337         VARIABLE BANK:
338              Node<T> - Node, holds the data item and reference to the next
339                              node in the stack.
340         DESCRIPTION:
341              This class is an implementation of ADT Stack.
342         ---------------------------------------------------------------------
343          */
344
345         //Reference based implementation
346         private Node<T> top;

347         public Stack() {
348              //Creates an empty stack.
349              createStack();
350         }

351         public void createStack() {
```

```
352             //Creates a new, empty stack.
353             top = null;
354         }

355         public boolean isEmpty() {
356             //Return true if stack is empty,
357             //false if not.
358             return top == null;
359         }
360
361         public void push(T newItem) {
362             //Adds newItem to the top of the stack.
363             top = new Node<T>(newItem, top);
364         }

365         public T pop() {
366             //Retrieves and then removes the top of the stack (the
367             //item that was added most recently).
368             if(!isEmpty()) {
369                 Node<T> temp = top;
370                 top = top.next;
371                 return temp.item;
372             }

373             return null; //remove
374         }

375         public void popAll() {
376             //Removes all items from the stack.
377             top = null;
378         }

379         public T peek() {
380             //Retrieves the top of the stack. That is, peek
381             //retrieves the item that was added most recently.
382             //Retrieval does not change the stack.

383             if(!isEmpty()) {
384                 return top.item;
385             }
386
387             System.out.println("Error: Cannot peek from empty stack.");
```

```
   388               return null;//remove
   389          }
   390     }

   391     class Node<T> {
   392          /*
   393          -----------------------------------------------------------------------
   394          CLASS NAME: Node
   395          VARIABLE BANK:
   396               item - T, data item contained by the node.
   397               Node<T> - next, reference to the next node.
   398          DESCRIPTION:
   399               This class is the Node used by the ADT Stack class. It allows for
   400               a single data item and a reference to the next node.
   401          -----------------------------------------------------------------------
   402           */
   403
   404          T item;
   405          Node<T> next;

   406          public Node() {
   407               //Create a default node.
   408               next = null;
   409               item = null;
   410          }

   411          public Node(T item, Node<T> next) {
   412               //Create a Node with the values passed.
   413               this.next = next;
   414               this.item = item;
   415          }
   416     }

:                              #null command
:
:
javac p6.java           #compile the java file
java p6                    #execute the file from the current directory
S-1
S1-
-S1
Result: 12.0000
```

```
C*S+S*0
CS*S0*+
+*CS*S0
Result: 260.0000

((((5-S)*2)-C)/3)
5S-2*C-3/
/-*-5S2C3
Result: -12.0000

3
3
3
Result: 3.0000

((((E-Q))))
EQ-
-EQ
Result: 1.0000

Y+0*Y*Y
Y0YY**+
+Y*0*YY
Result: 18.0000

R-1
R1-
-R1
Result: 1.0000

(Y/(5+(B-(2*4))))
Y5B24*-+/
/Y+5-B*24
Result: 9.0000

6-2+4/C-1*R-4
624C/1R*4--+-
-6+2-/4C-*1R4
Result: 1.8000

P*B
PB*
*PB
```

```
Result: 160.0000

(((P*4)/(A-S))/((2+6)-(3*2)))
P4*AS-/26+32*-/
//*P4-AS-+26*32
Result: -21.3333

(W)
W
W
Result: 10.0000

(((A-B)*((C+Q)/R))+E*1)
AB-CQ+R/*E1*+
+*-AB/+CQR*E1
Result: 79.0000

((((A+B)*C)-Y)/R)
AB+C*Y-R/
/-*+ABCYR
Result: 141.0000

(6/3+(C-(C*1)))
63/CC1*-+
+/63-C*C1
Result: 2.0000

(6/3+(C-(C*R))^2)
63/CCR*-2^+
+/63^-C*CR2
Result: 402.0000

R^D^2
RD2^^
^R^D2
Result: 512.0000

:
:
:
date                          #print the date
Mon Nov 11 07:25:11 CST 2013
```