

```

1 /**
2  * Starter code to implement an ExpressionParser. Your parser methods should
3  * use the following grammar:
4  * E := A | X
5  * A := A+M | M
6  * M := M*M | X
7  * X := (E) | L
8  * L := [0-9]+ | [a-z]
9  */
10 import java.lang.*;
11 public class SimpleExpressionParser implements ExpressionParser {
12     /**
13      * Attempts to create an expression tree -- flattened as much as possible --
14      * from the specified String.
15      * Throws a ExpressionParseException if the specified string cannot
16      * be parsed.
17      * @param str the string to parse into an expression tree
18      * @param withJavaFXControls you can just ignore this variable for R1
19      * @return the Expression object representing the parsed expression tree
20      */
21     public Expression parse (String str, boolean withJavaFXControls) throws
22     ExpressionParseException {
23         // Remove spaces -- this simplifies the parsing logic
24         str = str.replaceAll(" ", "");
25         Expression expression = parseExpression(str);
26
27         if (expression == null) {
28             // If we couldn't parse the string, then raise an error
29             throw new ExpressionParseException("Cannot parse expression: " + str);
30         }
31
32         // Flatten the expression before returning
33         expression.flatten();
34         return expression;
35     }
36
37     /**
38      *
39      * @param str input string to test for A | X
40      * @return the parsed expression or null
41      * @return the parsed expression or null
42      * @throws ExpressionParseException
43      */
44     private Expression parseE(String str) throws ExpressionParseException{
45         if(parseA(str) != null){
46             return parseA(str);
47         }
48         else if (parseX(str) != null){
49             return parseX(str);
50         }
51         return null;
52     }
53
54     /**
55      *
56      * @param str input string to test for A + M | M
57      * @return parsed expression or null
58      * @throws ExpressionParseException

```

```

57     */
58     private Expression parseA(String str) throws ExpressionParseException{
59         for (int i = 1; i < str.length() - 1; i++){
60             if(str.charAt(i) == '+' && parseA(str.substring(0,i)) != null &&
parseM(str.substring(i+1)) != null){
61                 final OperationExpression exp = new
OperationExpression(str.substring(i,i+1));
62                 final Expression child1 = parseA(str.substring(0,i));
63                 final Expression child2 = parseM(str.substring(i+1));
64                 exp.addSubexpression(child1);
65                 child1.setParent(exp);
66                 exp.addSubexpression(child2);
67                 child2.setParent(exp);
68                 return exp;
69             }
70         }
71         if(parseM(str) != null){
72             return parseM(str);
73         }
74         return null;
75     }
76
77     /**
78     *
79     * @param str string input string to test for (E) | L
80     * @return parsed expression or null
81     * @throws ExpressionParseException
82     */
83     private Expression parseX(String str) throws ExpressionParseException{
84
85         if(str.length() > 0 && str.charAt(0) == '(' && str.charAt(str.length()-1)
== ')'){
86             final OperationExpression exp = new OperationExpression("(");
87             if(parseE(str.substring(1,str.length()-1)) != null) {
88                 final Expression child1 = parseE(str.substring(1, str.length() - 1));
89                 exp.addSubexpression(child1);
90                 child1.setParent(exp);
91                 return exp;
92             }
93         }
94         else if (parseL(str) != null){
95             return parseL(str);
96         }
97         return null;
98     }
99
100    /**
101    *
102    * @param str string input to test if it is A-Z or 0-9
103    * @return parsed expression or null
104    * @throws ExpressionParseException
105    */
106    private Expression parseL(String str) throws ExpressionParseException{
107        if(str.length() == 1 && Character.isLetter(str.charAt(0)) ){
108            return new LiteralExpression(str);
109        }
110        try{
111            if(Integer.parseInt(str) >= 0 && str.charAt(0) != '+' && str.charAt(0)
!= '-') {
112                return new LiteralExpression(str);

```

```

113     }
114
115     }
116     catch(NumberFormatException e){
117         return null;
118     }
119     return null;
120
121 }
122 private Expression parseM(String str) throws ExpressionParseException{
123     for (int i = 1; i < str.length() - 1; i++){
124         if(str.charAt(i) == '*' && parseM(str.substring(0,i)) != null &&
parseM(str.substring(i+1)) != null){
125             final OperationExpression exp = new
OperationExpression(str.substring(i,i+1));
126             final Expression child1 = parseA(str.substring(0,i));
127             final Expression child2 = parseM(str.substring(i+1));
128             exp.addSubexpression(child1);
129             exp.addSubexpression(child2);
130             child1.setParent(exp);
131             child2.setParent(exp);
132             return exp;
133         }
134     }
135     if(parseX(str) != null){
136         return parseX(str);
137     }
138     return null;
139 }
140 protected Expression parseExpression (String str) throws
ExpressionParseException {
141     Expression expression;
142     expression = parseE(str);
143     // if string does follow a production rule then return the parsed tree
144     if(expression != null){
145         return expression;
146     }
147     return null;
148
149 }
150 }
151

```