```java
/**
 * Starter code to implement an ExpressionParser. Your parser methods should use the following grammar:
 * E := A | X
 * A := A+M | M
 * M := M*M | X
 * X := (E) | L
 * L := [0-9]+ | [a-z]
 */
import java.lang.*;
public class SimpleExpressionParser implements ExpressionParser {
  /*
   * Attempts to create an expression tree -- flattened as much as possible -- from the specified String.
         * Throws a ExpressionParseException if the specified string cannot be parsed.
   * @param str the string to parse into an expression tree
   * @param withJavaFXControls you can just ignore this variable for R1
   * @return the Expression object representing the parsed expression tree
   */
  public Expression parse (String str, boolean withJavaFXControls) throws ExpressionParseException {
    // Remove spaces -- this simplifies the parsing logic
    str = str.replaceAll(" ", "");
    Expression expression = parseExpression(str);

    if (expression == null) {
      // If we couldn't parse the string, then raise an error
      throw new ExpressionParseException("Cannot parse expression: " + str);
    }


    // Flatten the expression before returning
    expression.flatten();
    return expression;
  }

  /**
   *
   * @param str input string to test for A | X
   * @return the parsed expression or null
   * @return the parsed expression or null
   * @throws ExpressionParseException
   */
  private Expression parseE(String str) throws ExpressionParseException{
    if(parseA(str) != null){
      return parseA(str);
    }
    else if (parseX(str) != null){
      return parseX(str);
    }
    return null;

  }

  /**
   *
   * @param str input string to test for A + M | M
   * @return parsed expression or null
   * @throws ExpressionParseException
```

```
   */
  private Expression parseA(String str) throws ExpressionParseException{
    for (int i = 1; i < str.length() - 1; i++){
      if(str.charAt(i) == '+' && parseA(str.substring(0,i)) != null &&
parseM(str.substring(i+1)) != null){
        final OperationExpression exp = new
OperationExpression(str.substring(i,i+1));
        final Expression child1 = parseA(str.substring(0,i));
        final Expression child2 = parseM(str.substring(i+1));
        exp.addSubexpression(child1);
        child1.setParent(exp);
        exp.addSubexpression(child2);
        child2.setParent(exp);
        return exp;
      }
    }
    if(parseM(str) != null){
      return parseM(str);
    }
    return null;
  }

  /**
   *
   * @param str string input string to test for (E) | L
   * @return parsed expression or null
   * @throws ExpressionParseException
   */
  private Expression parseX(String str) throws ExpressionParseException{

    if(str.length() > 0 && str.charAt(0) == '(' && str.charAt(str.length()-1)
== ')'){
      final OperationExpression exp = new OperationExpression("()");
      if(parseE(str.substring(1,str.length()-1)) != null) {
        final Expression child1 = parseE(str.substring(1, str.length() - 1));
        exp.addSubexpression(child1);
        child1.setParent(exp);
        return exp;
      }
    }
    else if (parseL(str) != null){
      return parseL(str);
    }
    return null;
  }

  /**
   *
   * @param str string input to test if it is A-Z or 0-9
   * @return parsed expression or null
   * @throws ExpressionParseException
   */
  private Expression parseL(String str) throws ExpressionParseException{
    if(str.length() == 1 && Character.isLetter(str.charAt(0)) ){
      return new LiteralExpression(str);
    }
    try{
      if(Integer.parseInt(str) >= 0 && str.charAt(0) != '+' && str.charAt(0)
!= '-'){
        return new LiteralExpression(str);
```

```
      }

    }
    catch(NumberFormatException e){
      return null;
    }
    return null;

  }
  private Expression parseM(String str) throws ExpressionParseException{
    for (int i = 1; i < str.length() - 1; i++){
      if(str.charAt(i) == '*' && parseM(str.substring(0,i)) != null &&
parseM(str.substring(i+1)) != null){
        final OperationExpression exp = new
OperationExpression(str.substring(i,i+1));
        final Expression child1 = parseA(str.substring(0,i));
        final Expression child2 = parseM(str.substring(i+1));
        exp.addSubexpression(child1);
        exp.addSubexpression(child2);
        child1.setParent(exp);
        child2.setParent(exp);
        return exp;
      }
    }
    if(parseX(str) != null){
      return parseX(str);
    }
    return null;
  }
  protected Expression parseExpression (String str) throws
ExpressionParseException {
    Expression expression;
    expression = parseE(str);
      // if string does follow a production rule then return the parsed tree
      if(expression != null){
        return expression;
      }
      return null;

  }
}

```