```java
package jlisp;

import java.util.Scanner;

/**
 * COSC 2P03 Assignment 1
 *
 * Created by Matt Laidman on September 10, 2014.
 * Student Number 5199807
 *
 * This interpreter will allow you to set and call LISP data types.
 * Can set literally, but referencing other variables does not work :(
 *
 * ****************** Valid Input *********************
 *
 * setq var '(atomic (atomic2 atomic3) atomic4)
 * print var
 * print
 * stop
 *
 * ****************** Global Variables ******************
 *
 * reserved     Set of reserved words (lisp commands)
 * command      Entered LISP command
 * var          Variable to be set/printed
 * statement    Structure to store
 * literal      If statement was literal
 * myLISP       Stored LISPs
 *
 * @author Matt Laidman (5199807)
 * @version 1.0 (September 19, 2014)
 */

public class jLISP {

    private String[] reserved = {"setq", "print"};
    private String command, var, statement;
    private boolean literal;
    private Node myLISP;

    public jLISP ( ) {
        String in = "";
        command = "";
        while (!in.equalsIgnoreCase("stop")) {                  // Main input loop, "stop" quits
            in = new Scanner(System.in).nextLine();              // Get input
            if (!in.equalsIgnoreCase("stop")) {
                parse(in);                              // Convert input to command/var/literal/
                if (!isValid()) {                       // Check valid command and parentheses
                    System.out.println("ERR - Invalid Input");      // statement variables
                } else {
                    interpret();                        // Interpret command
                }
            }
        }
    }

    /**
     * Method determines if command entered is setq or print and
     * calls the appropriate functions.
     */

    private void interpret ( ) {
        Node ptr = myLISP;
        if (command.equalsIgnoreCase("setq")) {                  // If command is setq
            if (var.equals("") || statement.equals("")) {        // No statement or var given
                System.out.println("ERR - No statement/var to set");
            } else {
                if (varExists(var)) {                            // Check if var already exists
                    getNode(var).down = null;                    // Clear subtree if it does
                } else {
                    addVar(var);                                 // Create var if it does
                }
                setq(getNode(var));                              // Call setq
            }
        } else if (command.equalsIgnoreCase("print")){           // If command is print
            if (var.equals("")) {                                // If print all
                while (ptr.right != null) {
                    statement = ptr.data + " = (";
                    statement = print(ptr.down);                 // Call print
                    System.out.println(statement);
                    ptr = ptr.right;
                }
                statement = ptr.data + " = (";
                statement = print(ptr.down);
                System.out.println(statement);
            } else if (varExists(var)) {                         // If print single var and var exists
                statement = ptr.data + " = (";
                statement = print(getNode(var).down);            // Call print
                System.out.println(statement);
            } else {
                System.out.println("ERR - No such variable");    // If var given does not exist
            }
        }
    }

    /**
     * Recursively builds the LISP data structure from the given input statement
     *
     * @param ptr      The current Node to set
     */

    @SuppressWarnings("unchecked")
    private void setq (Node ptr) {
        if (statement.length() == 0) {                           // If no statement, return
            return;
        }

        statement = trim(statement);                             // Trim statement (remove leading spaces)
        if (statement.length() > 0 && statement.charAt(0) == '(') {   // If opening parenthesis, build down
            ptr.down = new Node(null, null, null);
            statement = statement.substring(1);
            setq(ptr.down);                                      // Call self with down Node
        }
        if (statement.length() > 0 && statement.charAt(0) == ')') {    // If closing parenthesis, return
            statement = statement.substring(1);
            return;
        }
        if (statement.length() > 0 && statement.charAt(0) != ')') {    // If element, build right
            if (!literal && varExists(getElem(statement))) {         // If not literal and var exists
                ptr.right = getNode(getElem(statement));
                statement = statement.substring(1);
                setq(ptr.right);                             // Call self with right Node
            } else {                                         // If not literal, or literal and var does not
                ptr.right = new Node(getElem(statement), null, null);  // exist
                statement = statement.substring(1);
                setq(ptr.right);                             // Call self with right Node.
            }
        }
    }

    /**
     * Recursively adds each element in the tree to the statement variable
     * to be returned and printed to the console.
     *
     * @param lisp     The var to print
     * @return       The string representation of the LISP
     */

    private String print (Node lisp) {
        if (lisp.data != null) {                             // If Node contains element
            if (statement.charAt(statement.length()-2) == ')') {      // If previous char in statement is ')'
                statement = statement + " ";                         // Add space
            }
            statement = statement + lisp.data + " ";                 // Add element to statement
        }
        if (lisp.right == null && lisp.data != null) {              // If end of list/sublist
            if (!lisp.data.equals(var)) {                     // If not parent var
                if (statement.charAt(statement.length()-1) == ' ') {   // If space previous, remove it
                    statement = statement.substring(0, statement.length()-1);
                }
                statement = statement + ")";                       // Add ')' to statement
            }
        }
        if (lisp.down != null) {                              // If sublist
            statement = statement + "(";                           // Add '(' to statement
            print(lisp.down);                                // Call self with down Node
        }
        if (lisp.right != null) {                             // If there is another element
            print(lisp.right);                              // Call self with right Node
        }
        return statement;                                    // Return completed statement
    }

    /**
     * Parses the given input into the command, var, literal, and setq commands
     *
     * @param in     The input to parse
     */

    private void parse (String in) {
        char[] inA = in.toCharArray();                  // Convert input to array
        command = "";                           // Clear variables
        var = "";
        statement = "";
        literal = false;
        int i = 0;
        while (i < inA.length && inA[i] != ' ') {           // Get command from first piece of input
            command = command + inA[i];
            i++;
        }
        i++;
        while (i < inA.length && inA[i] != ' ') {           // Get var from second piece
            var = var + inA[i];
            i++;
        }
        i++;
        if (i < inA.length && inA[i] == '\'') {           // Get if statement is literal
            literal = true;
            i++;
        }
        while (i < inA.length) {                    // Get statement from rest of input
            statement = statement + inA[i];
            i++;
        }
    }

    /**
     * Gets the next element to be set from statement
     *
     * @param in     The current statement variable to get element from
     * @return     The next element to be set
     */

    private String getElem (String in) {
        String next = "";
        char[] inA = in.toCharArray();                  // Convert in to array
        int i = 0;

        while (i < inA.length && inA[i] != ' ' && inA[i] != '(' && inA[i] != ')') {
            next = next + inA[i];                   // While char is still part of element (valid
            i++;                                // char)
        }
        return next;                            // Return next element
    }

    /**
     * Adds var to the LISP at the end of the LISP
     *
     * @param v     The element to be added
     */

    @SuppressWarnings("unchecked")
```

```java
private void addVar (String v) {
    Node ptr = myLISP;
    if (ptr == null) {                          // If no items in list
        myLISP = new Node(v, null, null);                   // Add at front
    } else {
        while (ptr.right != null) {                    // While there is a pointer
            ptr = ptr.right;                    // Get next pointer
        }
        ptr.right = new Node(v, null, null);                // Add v to end of list
    }
}

/**
 * Returns the Node of a given var if it exists
 *
 * @param v      The var to find
 * @return       The Node containing v
 */

private Node getNode (String v) {
    Node ptr = myLISP;
    if (ptr == null) {                          // If empty list
        return null;
    }
    while (ptr.right != null) {                    // While a var to the right exists
        if (ptr.data.equals(v)) {                 // If Node contains v
            return ptr;
        }
        ptr = ptr.right;                        // Get next Node
    }
    if (ptr.data.equals(v)) {
        return ptr;
    } else {
        return null;
    }
}

/**
 * Checks if the given var already exists in the list
 *
 * @param v      The var to check
 * @return       True if var is found, false if not
 */

private boolean varExists (String v) {
    Node ptr = myLISP;
    if (ptr == null) {                          // If empty list
        return false;
    }
    while (ptr.right != null) {                    // While list has nor to the right
        if (ptr.data.equals(v)) {                 // if Node contains v
            return true;
        }
        ptr = ptr.right;
    }
    return ptr.data.equals(v);
}

```

```java
    /**
     * Returns the given string will all leading spaces removed.
     *
     * @param in       The string to trim
     * @return         The trimmed string
     */

    private String trim (String in) {
        while (in.charAt(0) == ' ') {                       // While char is a space
            in = in.substring(1);                           // Remove first char
        }
        return in;
    }

    /**
     * Checks to see if entered command is a valid LISP command, and checks that the parentheses are valid.
     *
     * @return         True is validCMD and validBraces return true
     */

    private boolean isValid ( ) {
        return (validCMD() && validBraces());
    }

    /**
     * Checks to ensure that the given statement has an equal number of opening and closing parentheses
     * and that they match up.
     *
     * @return         True if given statement has valid parentheses
     */

    private boolean validBraces ( ) {
        if (command.equalsIgnoreCase("print") && statement.equalsIgnoreCase("")) {
            return true;
        }
        int count = 0;
        for (char c : statement.toCharArray()) {            // For each character in statement
            if (c == '(') {                                 // Increase count if opening
                count++;
            }
            if (c == ')') {                                 // Decrease count if closing
                count--;
            }
            if (count < 0) {                                // If too many closing parentheses
                return false;
            }
        }
        return (count == 0);                                // Return true if equal number of parentheses
    }

    /**
     * Checks the given command against the reserved words to ensure a valid command was entered
     *
     * @return       True if a valid command was entered
     */

    private boolean validCMD () {
        for (String s : reserved) {                   // For each String in reserved array
            if (command.equals(s)) {                   // If command equals reserved word
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {new jLISP();}
}
```