

This lab has **six (6)** checkpoints.

Learning Outcomes

Upon completing this lab, students should be able to

- **Trace** recursive text-reading code.
- **Implement** nodes for an Expression tree using JUnit tests to *define* what the nodes should do.
- **Add** a level to a recursive-descent parser.

Introduction

1. Consider the following recursive function, yep:

```
String yep(String str) {  
    String rv = "";  
    if (!str.isEmpty()) {  
        rv = str.substring(0, 1);  
        rv = yep(str.substring(1)) + rv;  
    }  
    return rv;  
}
```

What is the output of the following calls to yep?

```
System.out.println(yep("triceratops is the best dinosaur"));  
System.out.println(yep("amanaplanacanalpanama"));  
System.out.println(yep("hturt eht ecaf"));
```

✓ Show the lab instructor the output. Be prepared to explain how you figured it out.

2. What will the calls to xyz given below print?

```
String xyz(String digits, String other) {  
    if (digits.isEmpty() || (digits.length() != other.length()))  
        return "";  
  
    String result = "";  
    char d = digits.charAt(0);  
    char o = other.charAt(0);  
  
    int number = d - '0';  
    if (number > 0) {  
        d--;  
        result += o;  
        digits = d + digits.substring(1);  
    } else {  
        digits = digits.substring(1);  
        other = other.substring(1);  
    }  
    result += xyz(digits, other);  
    return result;  
}
```

```
System.out.println(xyz("123","abc"));
System.out.println(xyz("123","abcdefg"));
System.out.println(xyz("3210123","bokqjna"));
System.out.println(xyz("1771","+-+"));
```

✓ Show the lab instructor the output. Be prepared to explain how you figured it out.

3. Take the sample code (src folder) as a start. It is a Gradle project with a main and a test set up.

It will **not** compile because `expression/NumericExpression.java` and `expression/BinaryOperator.java` are both missing.

The interface, `expression/Expression.java` must be implemented by each of them. The three required methods are described for each class below:

`NumericExpression`

Constructor Takes a string containing an int value. Needs to keep the integer value.

value Returns the stored int.

precedence Returns a very **high** value (higher than any operator's precedence).

toString Returns string representation of the int value.

`BinaryOperatorExpression`

Constructor Takes *three* parameters: *Expression* for the *left-hand side* of the operator; *String* for the *operator* itself; another *Expression* for the *right-hand side* of the operator. Needs to keep all three values (all are `final`).

value Returns the operator applied to the values of the *lhs* and *rhs* Expressions. So, if op is "-", return left value minus right value.

precedence Return 10 for addition operators and 20 for multiply operations.

toString Returns string representation of *lhs*, spaces and operator, and string representation of *rhs*. Note: if precedence of either side is **lower** than this operator's precedence, wrap that side in parentheses.

As shipped, the code will not compile.

- (a) Write two classes, `NumericExpression.java` and `BinaryOperatorExpression.java`. Each implements the `Expression.java` interface. Implement stub functions so that Gradle can build the code. Get it to compile.

✓ Show the lab instructor compiling code.

- (b) Implement `NumericExpression` in `expression.NumericExpression.java`. Turn on the "numeric" tests in `build.gradle` to get everything but `toString` tested.

✓ Show the lab instructor compiling and testing code.

- (c) Implement `BinaryOperatorExpression` in `expression.BinaryOperatorExpression.java`. Turn on the "binop" tests in `build.gradle` to get everything but `toString` tested.

✓ Show the lab instructor compiling and testing code.

4. Add test cases for the `toString` methods of your two new classes in `TestExpressionHierarchy`. Use appropriate tags.

✓ Show the lab instructor compiling and testing code. Be prepared to discuss what you are testing (especially for `BinaryOperatorExpression.toString`) and why you think you are testing *enough*.