

Project 2

Please read the entire document carefully! There are multiple parts to this assignment. **Make sure to do all of them!**

1 Introduction

For this project, you will implement a program to get a taste of how calculators (and programming languages) can understand mathematical text.

Notes:

- To break up a string into “tokens”, consider using the `String.split` method (*e.g. to split a String input into an array, you can do `input.split("\\s+")`*).
- To convert a token (string) into a number (double), consider using `try/catch` blocks along with the `Double.parseDouble` function.
- Your functions will support the mathematical operations of `+`, `-`, `*`, `/`, and `^`.
- That last one, `^`, means exponentiation. It has higher precedence than the other operators (for example, $2 * 4 ^ 3 == 2 * 64 == 128$) and is right-associative (for example, $2 ^ 3 ^ 2 == 2 ^ (3 ^ 2) == 2 ^ 9 == 512$). (Note that `^` is not how Java does exponentiation. That’s OK! This is our calculator, we can do what we want!)
- In order to do exponents in Java, we can use `Math.pow`, e.g., 2^9 would be `Math.pow(2, 9)`.
- All other operators are left-associative (for example, $2 * 3 / 4 == (2 * 3) / 4 == 6 / 4 == 1.5$).

2 Stack

The first thing we’re going to have to do is implement a stack. We could use either an array-based structure, or we could use a link-based structure. While they both do work, for this assignment we’re going to implement an array-based stack.

Most of this is possibly already completed for a lab, but there are a few details that are different, so read carefully.

2.1 Generic Array-Based Structures

One *major* change will be that this stack implementation needs to be generic to support our use cases below (we will have at least two different types of stack). There is an interesting quirk in Java making generic array-based data structures, Java doesn’t let you make generic arrays! This means that the array inside the stack will need to be an array of `Objects`. This also means that whenever you take data out of the array (i.e., in `pop` and `peek` below), you will need to cast the result to be of the generic type, e.g.

```
// Assuming this.array is an array of objects and assuming that your generic
// type is named T.
T myVariable = (T) this.array[index];
```

Java will then give you a warning for an unchecked cast which we should suppress by placing the following code directly above the header of the methods that do the casting.

```
@SuppressWarnings("unchecked")
```

2.2 Implementation Details

In a file called `ArrayStack.java`, you will implement these methods for your generic `ArrayStack` class and you will write tests in `ArrayStackTest.java`.

- `emptyStack`: This *static* method will take no arguments and return an empty stack.
- `push`: This method will take a value of type `T` and place the value on the “top” of the stack. This method should mutate the given stack and won’t return anything.
- `pop`: This method takes no arguments and removes (and returns) the item at the “top” of the stack. If the stack is empty, then this method should throw an `IndexOutOfBoundsException`.
- `peek`: This method takes no arguments and returns (without removing) the item on the “top” of the stack. If the stack is empty, then this operation should raise an `IndexOutOfBoundsException`.
- `isEmpty`: This method takes no arguments and returns whether or not the stack is empty.
- `size`: This method takes no arguments and returns the number of items in the stack.

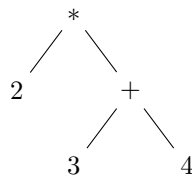
These should *all* be $O(1)$ (i.e., constant time) operations.

3 AST

Text is a little hard to work with in code. If we want to implement a calculator that takes its input as a string, a possible first step in this process could be to convert that string into a more convenient data structure with which to work. Consider the mathematical expression

$$2 * (3 + 4).$$

We could convert that into a tree like structure that would look like



To represent this tree in code, this means we will want two kinds of tree nodes, nodes that represent operations (e.g., `+`, `*`, etc.) and nodes that represent numbers. This kind of tree used to represent the syntax of a mathematical expression is called an Abstract Syntax Tree or AST for short.

In a file called `AST.java`, you will write an `AST` interface and in `NumNode.java` and `BinopNode.java` you will implement two classes (*which will likely be records*) that implement it `NumNode` and `BinopNode`. The `NumNode` class should contain a single `double` value, and the `BinopNode` class should contain two `AST` children, a left and a right.

For now, that’s all our `AST` will have, we’ll add more to it later.

4 Parsing a Postfix (RPN) Expression

4.1 Algorithm

In a file called `Parser.java`, you will implement this algorithm as a static method called `parsePostfix`. This method will take as input a single string and return an `AST`. Tests for `parsePostfix` should go in `ParserTest.java`.

While RPN will look strange until you are familiar with it, here you can begin to see some of its advantages for programmers. One such advantage of RPN is that it removes the need for parentheses. Infix notation supports operator precedence (`*` and `/` have higher precedence than `+` and `-`) and thus needs parentheses to override this precedence. This makes parsing such expressions much more difficult (see next section). RPN has no notion of precedence, the operators are processed in the order they are encountered. This makes evaluating RPN expressions fairly straightforward and is a perfect application for a stack data structure, we just follow these steps:

- Process the split string from left to right

- When a value is encountered:
 - Push the value onto a stack of AST nodes as a `NumNode`
- When an operator is encountered, apply it to the two expressions on the top of the stack. By this I mean:
 - Pop two AST nodes from the stack
 - Construct a `BinopNode` with the given operator and the popped nodes as its children
 - Push the result back onto the stack
- Return the last value remaining on the stack

As mentioned above, you may (and should) use the Java string method `String.split` to separate the string into tokens.

I would strongly suggest a helper function to handle applying an operator to the stack of expressions. We will need this functionality again so moving it into a helper function will allow us to reuse that bit of code easily. In fact, I would strongly suggest having *lots* of helper functions. Code is a lot easier to read when it's written in lots of small digestible pieces.

4.2 Exceptions

Note that the RPN expression will not always be valid. Specifically, your function should throw a `IllegalArgumentException` with the following messages in the following conditions:

- "empty input" if the input string is an empty string
- "invalid token" if one of the tokens is neither a valid operand nor a valid operator (e.g., `2 a +`)
- "insufficient operands" if the expression does not contain sufficient operands (e.g., `2 +`)
- "too many operands" if the expression contains too many operands (e.g., `2 3 4 +`)

To raise an exception with a message, you will throw `IllegalArgumentException("your message here")`.

For some of these, it's not immediately obvious where in the algorithm this error will surface. Try to run through the algorithm by hand for some of the examples and see where we would expect it to fail.

5 Parsing Infix Expressions

5.1 Algorithm

Also in `Parser.java`, you will implement this algorithm as a static method called `parseInfix` (with more tests in `ParserTest.java`).

We can (and you will!) also use a stack to convert an infix expression to an RPN expression via the Shunting-yard algorithm.

In fact you will need two stacks! One for the expressions like above, but also one for the operations to be performed. A major difference between postfix and infix will be that in infix when we see an operator, we need to postpone performing it until we know both of the operands.

The steps are:

- Process the expression from left to right.
- When you encounter a value:
 - Push the value onto the expression stack as a `NumNode`
- When you encounter an opening parenthesis:
 - Push it onto the operators stack

- When you encounter a closing parenthesis:
 - Until the top of the operators stack is an opening parenthesis, pop operators off the operators stack apply them to the expressions stack (*note that apply here has the same meaning as above, perhaps reusing that helper function I suggested you write*)
 - Pop the opening parenthesis from the operators stack
- When you encounter an operator, o_1 :
 - While there is an operator, o_2 on the top of the operators stack and either:
 - * o_2 has greater precedence than o_1 , or
 - * they have the same precedence and o_1 is left-associative
 Pop o_2 from the operators stack and apply it to the expressions stack
 - Then push o_1 onto the stack
- When you get to the end of the infix expression, pop (and apply) all remaining operators.

The following table summarizes the operator precedence, from the highest precedence at the top to the lowest precedence at the bottom. Operators in the same line have the same precedence.

Operators	Notes
$\wedge$	Right associative
$*, /$	Left associative
$+, -$	Left associative

5.2 Exceptions

Note that the infix expression will not always be valid. Specifically, your function should throw an `IllegalArgumentException` with the following messages in the following conditions:

- "empty input" if the input string is an empty string
- "invalid token" if one of the tokens is neither a valid operand not a valid operator (e.g., `2 + a`)
- "insufficient operands" if the expression does not contains sufficient operands (e.g., `2 +`)
- "too many operands" if the expression contains too many operands (e.g., `2 + 3 4`)
- "mismatched open paren" if an open paren is never closed (e.g., `"( 2 + 3"`)
- "mismatched close paren" if a close paren was never opened (e.g., `"2 + 3 )"`)

To raise an exception with a message, you will throw `IllegalArgumentException("your message here")`.

Similar to the above, for some of these it's also not immediately obvious where in the algorithm this error will surface. Try to run though the algorithm by hand for some of the examples and see where we would expect it to fail.

6 Eval

Now that we have the mathematical expressions parsed into ASTs, we can write the code to actually evaluate them! This is where we will actually do the mathematical calculation.

Add a method to your AST interface called `eval`. Tests for `eval` will go in `ASTTest.java`. The `eval` method should interpret the given AST into a double. Now implement this method in both the `NumNode` class and the `BinopNode` class. Now finally, we should be able to test that:

```
parseInfix("2 * ( 3 + 4 )").eval()
```

should give us 14.

7 Testing

For each part, you are *required* to achieve 100% test coverage. This means that every line of code in your functions *must* be executed at some point in at least one of your tests.

8 Deliverables

To make sure you followed everything above correctly, by the time you're done you should have these files:

Implementation:

- `ArrayStack.java`
- `AST.java`
- `NumNode.java`
- `BinopNode.java`
- `Parser.java`

Tests:

- `ArrayStackTest.java`
- `ASTTest.java`
- `ParserTest.java`

9 Submission

You will submit your code to Gradescope. Details forthcoming.