# 6.005 Elements of Software Construction | Fall 2011
# Problem Set 3: Calculator Parser

This problem set explores parsing.

Much of the code that you write for problems in this problem set will depend heavily on how you decided to implement earlier parts of the problem set, so you may want to read through the entire assignment before jumping into writing code.

**Do not change the signatures or specifications of methods in the MultiUnitCalculator class, or any of the names of classes and packages that we have provided you. Your code will be tested automatically, and it will break our testing suite if you do so. You are free, however, to modify the specifications and signatures internal to the workings of your Lexer and Parser.**

## Overview

It's often convenient to use different units in the same computation. For example, to figure out how many lines of twelve-point type fit in a six-inch column, it would be nice to have a calculator that accepted the expression "6in/12pt". Construct two grammars for such a language: first, a lexical grammar that breaks the sequence of characters into numbers, unit specifiers, operators, and left/right parentheses, filtering out spaces and tabs (but not requiring them as delimiters); and second, a syntactic grammar that groups expressions appropriately.

Your grammar should be able to recognize the following types of expressions:

- Arithmetic expressions with +, -, *, and /. You can assume that the order of operations will *always* be made explicit with parentheses.
- Expressions with integer and decimal operands, both as scalars (numbers without units) and inches and points. Point operands will be the number followed by 'pt,' and inches will be followed by 'in.' The result of evaluating the expression should be in appropriate units also.

    - inches/scalar = inches
    - inches*scalar = inches
    - inches/inches = scalar
    - inches/points = scalar

    Assume similar combinations for substitutions of inches and points.

    The following combinations are not as intuitive (or even true to the real world), but we offer this specification to simplify the unit types you must output and allow for consistency in grading:

    - scalar/inches = inches
    - inches*inches = inches
    - scalar+inches = inches
    - inches+points = inches (use units of the first operand)
- Unit conversion expressions (also made explicit by parentheses), represented by 'in' or 'pt' following an expression.
- As said above, whitespace should be ignored.

You will be expected to be able evaluate all the following expressions, with the result presented after the '=':

- 3+2.4 = 5.4
- 3 + 2.4 = 5.4
- 1 - 2.4 = -1.4
- (3 + 4)*2.4 = 16.8
- 3 + 2.4in = 5.4in
- 3pt * 2.4in = 518.4pt
- 3in * 2.4 = 7.2in
- 4pt+(3in*2.4) = 522.4pt
- 4pt+((3in*2.4)) = 522.4pt
- (3 + 2.4) in = 5.4in
- (3in * 2.4) pt = 518.4pt

You will NOT be expected to evaluate the following expressions:

- 3+2+1 (order of operations not made explicit)
- -2+(3+4) (negative number in input)
- 3+-4 (negative number in input)
- 2+(3+4)+2 (order still not explicit)

- (3+4)^2 (do not need to support ^)
- 2+3in pt (order not explicit)
- 2()+3in (extraneous parens)

If your calculator receives an expression that it does not know how to handle, you should display some sort of error message, ideally with some information about why the input could not be evaluated, such as "Order of operations not made explicit," or "Cannot divide by 0."

We have provided you with some skeleton code for this calculator. It consists of a simple prompt loop, in the main method of MultiUnitCalculator, for users to enter expressions. There is also a Lexer class along with a Type class, and a Parser class for interpreting the input expression String and evaluating it. This problem set will walk you through the implementation of these classes.

## Problem 1: Specify a Grammar

Before jumping into the implementation, you should specify a grammar for your calculator. Your grammar should be able to recognize the expressions described in the **Overview** section above.

**a. [5 points]** Start by deciding what symbols or terminals your grammar will contain. This should include things like '+,' '6,' and 'pt.' List these in a comment at the top of Type.java.

**b. [5 points]** Next, group your symbols into Types that your Lexer will recognize, based on their function in an expression. For example, plus, divide, and open parenthese will be different Types, but you can specify just one Type for all numbers. Indicate your groupings in your comment in Type.java and then create a Type enum for each of your groups.

**c. [15 points]** Finally, write out your grammar in a comment at the top of Parser.java. Use the production syntax defined in lecture.

## Problem 2: Implement the Lexer

The first step for evaluating an expression will be converting the user input String into a set of tokens that your parser will be able to recognize. These tokens will be parts of the String paired with the Types that you defined in Problem 1. We've provided you with Lexer.java, which defines an internal Token class for you to utilize when implementing your Lexer. To actual recognize the tokens in the input Strings, you may want to use regular expressions, as

discussed in Lexer. Java provides support for regexes in the Java.util.regex package, as presented in lecture. Eventually you will pass the Lexer to the Parser constructor, but exactly how the Lexer presents the Tokens to the Parser is up to you. *Hint: You can think about the Lexer consuming the expression one character at a time and passing the resulting Tokens to the Parser, but this is only one way of doing things.*

**a. [5 points]** Write the spec for the Lexer constructor method, and for any other methods you expect to need for tokenization.

**b. [5 points]** Write test cases for these methods in a separate file.

**c. [10 points]** Now implement your Lexer class based on your specification. Test it using the tests that you wrote.

## Problem 3: Implement the Parser

Your parser should use the tokenization produced by the Lexer to parse the expression according to the grammar you defined in Problem 1. Exactly how you do this is up to you; the only thing that we enforce is that the Lexer is passed as an argument to the Parser constructor.

**a. [10 points]** Write the spec for the Parser constructor method, the evaluate method, and any other methods you expect to need for parsing and evaluating the tokenized expression.

**b. [5 points]** Write test cases for these methods in a separate file.

**c. [20 points]** Implement the Parser and make sure all your tests pass.

## Problem 4: Put it all together

Finally, apply your Lexer and Parser to the input expression as defined in the spec for evaluate in MultiUnitCalculator.java.

**a. [5 points]** First, write test cases for the evaluate method in CalculatorTest.java.

**c. [10 points]** Implement the evaluate method using your Lexer and Parser. *Hint: This shouldn't be too hard, or you might want to rethink how you wrote your Lexer and Parser.*

## Optional Extension

This extension is optional and will not affect your grade in any way. The only reason to try it is to quench your burning desire to program.

Play around some more with unit conversion. Add support for different units, like seconds or mph.

You might also want to experiment with evaluating expressions where the order of operations is not made explicit by parentheses, and your parser must resort to PEMDAS!

## Before You're Done...

Double check that you didn't change the signature of anything in the MultiUnitCalculator class, or change any package or class names.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

Try to make sure that your code conforms to standard Java naming conventions. That is, class names should be StartingUpperCamelCase, and variables and methods should be startingLowerCamelCase.

6.005 Elements of Software Construction

Fall 2011