CSE12 - Spring 2014 HW #8

Unicalc

(400 points)
Partner Declaration Due 5pm 23 May 2014
Checkpoint Solution Due 11:59pm 27 May 2014
Final Solution Due 11:59pm 3 June 2014

This assignment is based on the CS 60 "Unicalc" assignment at Harvey Mudd College.

This assignment is a partner-encouraged assignment. You may complete the assignment with one other student or individually. Your partner may be from either section. You must declare your partner on the partner form (linked from this Piazza post: https://piazza.com/class/hr79v506xgw3xy?cid=1032) by the deadline listed above. You and your partner may share code, and we encourage (but do not require) you to work in a pair-programming style, where you work together at one computer on the assignment. You will submit only one submission between the two of you. **Keep in mind that both members of the partnership are responsible for understanding and *being able to write* all the code that is required in this assignment.**

You may also ask Professors/TAs/Tutors for some guidance and help., but you can't copy code. You can discuss the assignment with your classmates who are not your partner, but don't look at or copy each others code or written answers.

We are providing the following files for this assignment:

- Tokenizer.java
- Unicalc.java
- AST.java
- QuantityDB.java

The rest you will create yourself. Note that most of these starter files will not compile until you at least have a stub implementation of your Quantity class.

You will submit the following file for this assignment:

- **Tokenizer.java (likely unchanged from the provided file)**
- **Quantity.java**
- **Unicalc.java**
- **AST.java**
- **QuantityTester.java (collected at the checkpoint and the final)**

- **UnicalcTester.java (collected at the checkpoint and the final)**
- **ASTTester.java (collected at the checkpoint and the final)**

At the top of each of your files in a comment, don't forget to include this information

```
CSE 12 Homework 8
Your Name and Your Partner's Name
Your PID and Your Partner's PID
Section [Your section] (A00 for Alvarado, B00 for Papadopoulos) for
both partners
The date
```

Overview: A calculator, with units: "Unicalc"

Unicalc is a calculator program that handles numerical quantities including physical and other units, e.g.,

420 seconds
4.20 meters / second
42.42 mile / hour + 99 km / day

Units are important to computation because they eliminate an element sometimes left to human interpretation. In the area of engineering, failure to interpret numbers without their units specified has been known to lead to failure of space missions. In the Wikipedia article about NASA's Mars Climate Orbiter, it is stated:

> "Twenty-four hours prior to orbital insertion, calculations placed the orbiter at an altitude of 110 kilometers; 80 kilometers is the minimum altitude that Mars Climate Orbiter was thought to be capable of surviving during this maneuver. Final calculations placed the spacecraft in a trajectory that would have taken the orbiter within 57 kilometers of the surface where the spacecraft likely disintegrated because of atmospheric stresses. The primary cause of this discrepancy was human error. Specifically, the flight system software on the Mars Climate Orbiter was written to calculate thruster performance using the metric unit Newtons (N), while the ground crew was entering course correction and thruster data using the Imperial measure Pound-force (lbf)."

A little closer to home, you have probably had to at some point in your past do calculations that included units, and it's quite likely that you will have to again at some point in your future. It is our hope that this calculator will prove useful in those classes.

In this assignment you will be building a program that interprets and evaluates this simple computer language to do unit conversions.

The Unicalc Interpreter

This section gives an overview of what a user's interaction with the Unicalc interpreter looks like. The Unicalc interpreter is the program you will write for the assignment. You can think of the Unicalc interpreter as being analogous to the Interactions Pane in Dr. Java, except instead of interpreting statements in Java, it interprets statements in the Unicalc language. When the user types a legal Unicalc statement at the input> prompt, the Unicalc interpreter returns and displays the result of evaluating that expression.

```
> java Unicalc

input>  14m+9m
Tokens: [14, m, +, 9, m]
AST: Sum(Product(Value(14.0),Value(1.0
m)),Product(Value(9.0),Value(1.0 m)))

Result: 23.0 m

input>  60 Hz * 30s
Tokens: [60, Hz, *, 30, s]
AST: Product(Product(Value(60.0),Value(1.0
Hz)),Product(Value(30.0),Value(1.0 s)))

Result: 1800.0 Hz s

input>  # 60Hz * 30s
Tokens: [#, 60, Hz, *, 30, s]
AST: Normalize(Product(Product(Value(60.0),Value(1.0
Hz)),Product(Value(30.0),Value(1.0 s))))

Result: 1800.0

input>  # 364.4 smoot
Tokens: [#, 364.4, smoot]
AST: Normalize(Product(Value(364.4),Value(1.0 smoot)))

Result: 364.4 smoot

input>  def smoot 67 in
Tokens: [def, smoot, 67, in]
AST: Define(smoot,Product(Value(67.0),Value(1.0 in)))
```

```
Result: 67.0 in

input>  # 364.4 smoot
Tokens: [#, 364.4, smoot]
AST: Normalize(Product(Value(364.4),Value(1.0 smoot)))

Result: 620.13592 meter
```

The bold emphasis has been added so that you can see the core interaction happening in the interpreter, and the user's input is shown in blue. The user types a line in "Unicalc language" to be evaluated at the input> prompt. The Unicalc interpreter then Tokenizes the line, parses it, evaluates it and prints the result. The language is described in more detail below, but here's an annotated version of the interaction above to help you understand what's going on. Our annotations are in green, and are not part of the output of the Unicalc interpreter:

```
> java Unicalc

input>  14m+9m   The user enters a unit calculation expression
Tokens: [14, m, +, 9, m]   The result of tokenizing the input string
AST: Sum(Product(Value(14.0),Value(1.0
m)),Product(Value(9.0),Value(1.0 m)))   The AST (split across several lines)

Result: 23.0 m   The result of evaluating the sum

input>  60 Hz * 30s
Tokens: [60, Hz, *, 30, s]
AST: Product(Product(Value(60.0),Value(1.0
Hz)),Product(Value(30.0),Value(1.0 s)))

Result: 1800.0 Hz s   Notice how the units don't cancel unless we explicitly
normalize the expression, as in the next example.

input>  # 60Hz * 30s
Tokens: [#, 60, Hz, *, 30, s]
AST: Normalize(Product(Product(Value(60.0),Value(1.0
Hz)),Product(Value(30.0),Value(1.0 s))))

Result: 1800.0   Because the user preceded the expression with #, the result
was normalized

input>  # 364.4 smoot
Tokens: [#, 364.4, smoot]
AST: Normalize(Product(Value(364.4),Value(1.0 smoot)))
```

Result: `364.4 smoot` If a unit is not in the database, its normalized value is just one of itself. So 364.4 smoots normalizes to 364.4 smoots

input> `def smoot 67 in` Add the unit "smoot" to the database of units. Define a smoot as 67 inches.

```
Tokens: [def, smoot, 67, in]
AST: Define(smoot,Product(Value(67.0),Value(1.0 in)))
```

Result: `67.0 in` The result of this expression is defined to be the value assigned to the variable (i.e. unit name).

input> `# 364.4 smoot`

```
Tokens: [#, 364.4, smoot]
AST: Normalize(Product(Value(364.4),Value(1.0 smoot)))
```

Result: `620.13592 meter` Now we can represent smoots in terms of the most basic unit of length in the database: meters.

Overview of the code you will write

As you progress in your CS career, you will take on more responsibility for the design of your objects, but we're not quite asking you to do that yet. In this section we give a high level overview for the classes that you will be writing. More details on each class are described below.

Note that some of these classes (or parts of some of these classes) are provided for you, but they will not compile until you at least create the classes they use including "stubs" of the methods they call.

Quantity.java

This class will represent the quantities with their units. E.g. "3.0 foot" or "4.2 m / s". This class will contain all of the methods to manipulate these quantities, including doing arithmetic with them and converting them into other units (e.g. feet to meters). You will create this class completely from scratch.

Unicalc.java

This class will contain methods for parsing a list of tokens according to the rules of the Unicalc language. The ultimate goal of these parsing methods to produce an Abstract Syntax Tree that can be passed to the evaluator. This class also contains a main method that implements the read-eval-print loop for the Unicalc interpreter. This is the class you run to run the Unicalc interpreter. We provide you with a starter version of this class, which you will add to.

AST.java
This class will contain methods necessary to represent and evaluate an Abstract Syntax Tree for the Unicalc language. We provide you with a starter version of this class, which you will add to.

QuantityTester.java, UnicalcTester.java, ASTTester.java
These are three JUnit tests classes that test their respective classes. These are the three classes that must be completed by and will be collected at the checkpoint deadline. You will create these classes completely from scratch.

Other provided files

Tokenizer.java
This class will contain a method that takes a raw input string from a user and returns a list of tokens that are relevant to the Unicalc language. This class is provided for you and does not need to be modified.

UnicalcDB.java
This class contains the starting database of unit conversions used by the Unicalc interpreter. It is provided for you and you do not need to modify it (but again, it will not initially compile).

Todo for the checkpoint: The three testers

As usual, we encourage a test-driven development process. Therefore, for the checkpoint deadline what you need to submit are the three tester classes (QuantityTester.java, UnicalcTester.java, ASTTester.java) with *at least* 1 meaningful test for each of the public methods in the class they are testing. You only need to write tests for the public methods that are either provided as incomplete in the starter code, or specified below. You do not need to write tests for any methods that you do not change or add to in the starter code. You also do not need to write tests for any private methods you write or any public methods that are not either described below or provided as part of the starter code.

Specifically, here are the methods you need at least one test for for in each class:
Quantity:

- each of the 3 constructors
- mul
- div
- add
- sub
- negate
- normalizedUnit

- normalize
- pow
- equals
- hashCode

Unicalc:

- S
- L
- E
- P
- K
- Q
- R

AST: One test for eval for each of the classes including:

- Product
- Quotient
- Sum
- Difference
- Power
- Negation
- Normalize
- Define
- Value

We will test your testers against a known working implementation of each class. You can do the same by testing against the .class files for the working solution we provide.

Note that it is necessary to completely understand what each of the methods for each class does, but it is not necessary to implement all the method before you write the tester. If you like, you can write all of your tester methods before writing any of the methods in the classes you are testing. You will be able to test your tester by running it against the class files with the solution that we provide.

However, although the three tester files are the first files that we will collect, they do not need to be completely finished before you start on the other classes. In fact, we *strongly* encourage you to develop at least a few of the methods in the classes as well. That way you'll have a bit more work done before the checkpoint. If you leave all the coding on the three main classes until after the checkpoint you'll have a tough second week. And it's more fun to write the actual code than the testing code, we think. :)

Finally, you can (and should!) continue to augment your testers after the checkpoint assignment as you develop your Quantity, Unicalc and AST classes.

Part 1: Representing the Quantities (Quantity.java)

Unlike in previous assignments, we will not be providing you any starter code or documentation for this class. The specification below describes exactly what you must implement in your Quantity.java class. Follow these specifications exactly, and be sure to include complete javadoc comments everywhere that is necessary.

An object of class `Quantity` should represent a numerical value having numerical value and attached units, e.g., “9.8 meters per second squared.” A `Quantity` object always contain a value (e.g. 9.8) and some units (e.g., meters per second squared), though in some cases the collection of units might be empty (for “dimensionless” numbers like π).

Start out by creating a file `Quantity.java` that contains a definition of a class `Quantity`. The `Quantity` class must contain at least the following fields:

- The numerical value itself (as a `double`)
- The attached units (as a `Map<String,Integer>`).

This last field requires a bit more explanation: the map* gives us the (non-zero) exponent for each attached unit. For example, a quantity measured in “meter per second squared” would have a map that associates the key "meter" with the exponent 1 and the key "second" associated with the exponent -2. Similarly, a quantity measured in kg m^2 / s^3 would have "kg" mapping to 1, and "m" mapping to 2, and "s" mapping to -3. Try to make sure this map never gives any unit name an exponent of 0. (Take that unit name out of the map, instead).

*Note that the Map API is covered in Ch 12 of the textbook and we will discuss it in class on Thursday or Friday of week 8. But don’t let that stop you from starting on the assignment before then! The specific Map data structure you will want to use here is Java’s HashMap, for which you can find documentation here. [http://docs.oracle.com/javase/7/docs/api/java/util/Map.html] Just read the documentation (and optionally the chapter in the book) and you’ll be good to go.

Your `Quantity` class must also support for the following methods:

Constructors

- A no-argument constructor that creates a default quantity of value 1 and no units. (Note that even when there are no units; the object should still contain a map object; it will just be a map with no entries.)
- A constructor that takes a single `Quantity` argument, and creates a deep copy. By deep copy here we mean that the new `Quantity` will be a different object with its own instance variables, and that it will contain a *copy* of the `Map` object referenced by the argument Quantity. Whether or not you copy

the `String` and `Integer` objects *inside* the Map does not matter since Integers and Strings are immutable.

- A 3-argument constructor: a `double` (the numeric value), a `List<String>` of the units in the numerator (i.e., with positive exponents), and a `List<String>` of the units in the denominator (i.e., the units with negative exponents). For example, we can use this constructor to represent the quantity 9.8 m/s^2

  ```
  new Quantity(9.8, Arrays.asList("m"), Arrays.asList("s", "s"))
  ```

  If either of the list arguments is null, this method should throw an `IllegalArgumentException`.

Other methods

- A method `mul` that takes a single Quantity argument, multiplies `this` by the argument, and returns the result. The returned value should be a **brand new** `Quantity` object; neither `this` quantity nor the argument quantity should change!

  This method should throw an `IllegalArgumentException` if its argument is null.

- A method `div` that takes a single `Quantity` argument, divides `this` by the argument, and returns the result. The returned value should be a **brand new** Quantity object; neither this quantity nor the argument quantity should change!

  This method should throw an IllegalArgumentException if its argument is null or if the value in the Quantity argument is zero.

- A method `pow` that takes a single `int` argument (positive, negative, or zero!), raises `this` to the given power, and returns the result. The result should be a **brand new** Quantity object; `this` quantity should not change!

- A method `add` that takes a single `Quantity` argument, adds `this` to it, and returns the result. The result should be a **brand new** Quantity object; neither this quantity nor the argument quantity should change!

  This method should throw an `IllegalArgumentException` if its argument is null or if the two Quantity objects do not have the same units.

  Note: When checking whether the quantities have the same units: the .equals(...) method for two maps should do what you want, as long as the

maps don't associate any unit with the zero exponent.

- A method `sub` that takes a single `Quantity` argument, subtracts it from `this`, and returns the result. The result should be a **brand new** Quantity object; neither this quantity nor the argument quantity should change!

  This method should throw an `IllegalArgumentException` if its argument is null or if the two Quantity objects do not have the same units.

- A method `negate` that takes no arguments and returns a new `Quantity` which is the negation of `this` Quantity. The result should be a **brand new** Quantity object; `this` quantity should not change!

- A method `toString()` that returns the quantity as a `String`. In fact, for consistency of formatting, we are providing this method for you. Please add the line:

  ```
  import java.text.DecimalFormat;
  ```

  to the top of your file, and then paste the following code into your `Quantity` class, and update the first few lines to match the names of the fields in a Quantity:

```
public String toString()
{
  // XXX You will need to fix these lines to match your fields!
  double valueOfTheQuantity = this.NAME_OF_RELEVANT_FIELD;
  Map<String,Integer> mapOfTheQuantity = this.NAME_OF_RELEVANT_FIELD;

  // Ensure we get the units in order
  TreeSet<String> orderedUnits =
      new TreeSet<String>(mapOfTheQuantity.keySet());

  StringBuffer unitsString = new StringBuffer();

  for (String key : orderedUnits) {
    int expt = mapOfTheQuantity.get(key);
    unitsString.append(" " + key);
    if (expt != 1)
        unitsString.append("^" + expt);
  }

  // Used to convert doubles to a string with a
  //   fixed maximum number of decimal places.
```

```
DecimalFormat df = new DecimalFormat("0.0####");

// Put it all together and return.
return df.format(valueOfTheQuantity)
         + unitsString.toString();
}
```

- A boolean method `equals` that takes any single `Object`, and returns `true` if and only if that object is a Quantity whose units are exactly the same as the calling object and whose value is the same when rounded to five places after the decimal point. E.g. the values 0.111112 and 0.111113 are considered equals while 0.111112 and 0.111118 are not.

  It is important that this method take an `Object` as an argument, and *not* a Quantity so that it overrides the equals method provided in the Object class. The Java operator `instanceof` will be useful here.

  You do not have to worry about equivalent quantities expressed in different units here. That is, a Quantity representing 100 cm is considered not equal to a Quantity representing 1 meter.

  Hint: An easy way to implement this method is to compare the String representation of the two objects once you've determined that the argument is in fact a Quantity object. The provided code for toString above will automatically round Quantities' values to the nearest 5 places after the decimal point.

- A method `hashCode()` that returns an integer, such that equal Quantities always return the same integer. (Hint: this.toString().hashCode())

The remaining methods we need in Quantity use a "database" telling us how to define some units in terms of others, e.g., a "day" is equivalent to exactly 24 hours. We represent this database using a `Map<String,Quantity>`. Units that appear as keys in this map are said to be "defined" units; units that are not keys in this map are said to be "primitive" units. A quantity is said to be **normalized** if the units it its map are all primitive. The definitions in the database might refer to other defined units (e.g., a day is 24 hours, an hour is 60 minutes, and a minute is 60 seconds).

Here is some code to create a very simple map for test purposes; feel free to include this (or an extension of this) in your `QuantityTester` class (and possibly your other tester classes:

```
Map<String,Quantity> db = new HashMap<String,Quantity>();
List<String> emp = new ArrayList<String>();

db.put("km", new Quantity(1000, Arrays.asList("meter"), emp));
db.put("day", new Quantity(24, Arrays.asList("hour"), emp));
db.put("hour", new Quantity(60, Arrays.asList("minute"), emp));
db.put("minute", new Quantity(60, Arrays.asList("second"), emp));
db.put("hertz", new Quantity(1, emp, Arrays.asList("second")));
db.put("kph", new Quantity(1, Arrays.asList("km"),
                           Arrays.asList("hour")));
```

(You will also need `import java.util.Map; import java.util.HashMap; import java.util.List; import java.util.ArrayList; import java.util.Arrays;` at the top of your file)

You may also use the much more extensive DB that we have provided in the file `QuantityDB.java` (which has one static method: getDB that returns the database).

Then add the following additional methods to your Quantity class:

- A static method `normalizedUnit` that takes a `String` (the name of a unit) and a `Map<String,Quantity>` (a units database). It should create a brand-new **normalized** `Quantity` equivalent to 1 of the given unit. For example, if db is the database defined above, then `Quantity.normalizedUnit("km", db)` should return the quantity `1000 meter`; `Quantity.normalizedUnit("hour", db)` should return the quantity `3600 second`; and `Quantity.normalizedUnit("kph",db)` would return the quantity `0.27777... meters per second`.

  If the unit does not appear as a key in the database then this method returns a new Quantity representing 1 of the argument unit. For example, if db is the database defined above then `Quantity.normalizedUnit( "smoot", db )` should return the quantity `1 smoot`.
- A (non-static) method `normalize` that takes in a database in the same format as above and returns a copy of `this` but in normalized form (with all defined units expanded out into primitive units). For example, normalizing the quantity 60 kph, given the above database, should produce the quantity 16.6666... meters per second.

Hints for normalizedUnit and normalize

The code for `normalizedUnit` and `normalize` is surprisingly simple, since each can use the other (in addition to `mul`, `div`, and `pow`).

For `normalizedUnit`, either the given unit is primitive (in which case the answer is easy to construct), or it has a definition in the database that could be normalized.

Similarly, to normalize a quantity like 30 km^2 / hour, it suffices to observe that

1. The products, quotients, and power of normalized quantities are normalized, and

2. 30 km^2/hour ==

   30 km^2/hour * (1000 meter / km)^2 * (3600 second/hour) ==

   30 km^2 hour^-1 * (1000 meter / km)^2 * (hour/3600 second)^-1

Look carefully at what's happening in step 2. 1000 meter / km is of course equal to 1 (since a km is 1000 meters). So multiplying
30 km^2/hour * (1000 meter / km) ^2 is equivalent to multiplying
30km^2/hour * 1, but in the former case the km units will cancel, leaving the result in meters instead of km. The quantities "1000 meter" and "3600 second" should look familiar from the discussion of normalizeUnit.

You should feel free to add additional constructors and additional (private) helper methods as you see fit. For example, we wrote a method `adjustExponentBy(String unitName, int delta)` that adds delta to the existing exponent in the units map. (It also adds the unit to the map, if it wasn't there already, and deletes it from the map, if the adjusted exponent is 0.) Then, the constructor that takes a list of units and just calls `adjustExponentBy(..., 1)` for each element in the numerator, and calls `adjustExponentBy(..., -1)` for each element in the denominator.

Part 2: The Parser (Unicalc.java)

You will find all of the parsing methods in the Unicalc class. Your calculator should accept any input that can be produced by (or, more importantly, recognized by) the following grammar:

```
S := def W L | L
L := # E | E
E := P + E | P - E | P
P := K * P | K / P | K
K := - K | Q
Q := R | R Q
R := V | V ^ J
V := D | W | ( L )
```

```
J := I | - I

I -> (any nonnegative integer)
D -> (any nonnegative number, integer or floating-point)
W -> (any word that is a unit name)
```

Once you have your Quantity class working, when you run the Unicalc main function the provided parsing code should already handle extremely simple inputs, including

- (single) positive numbers (e.g., 42 or 98.6)
- (single) unit names (e.g., meter or furlong)
- parentheses around any of the above. (e.g., (3) or ((meter)))

For this part, you need to implement the rest of the recursive-descent parser for the Unicalc language, according to the grammar shown above.

As discussed in class in Week 8, for each variable/nonterminal in the grammar there should be exactly one parsing method whose job is to consume a prefix of the remaining input, a prefix derivable from that nonterminal in the grammar. By peeking at the upcoming token or tokens (if necessary), each function should decide which production rule corresponds best to the input.

Development Strategy. We *strongly* suggest you start from the bottom of the grammar up. First start with R(), and make sure your parser can handle strings derivable from R in the grammar, like meter^2 or (3)^-1. Then get Q() working, and try it on strings derivable from Q in the grammar like 2 meter or 9.8 meter^2 second^-1. Then move on to K(), to P(), and so on.

Parser output. As discussed in class, we want the parser (and each parsing function) to construct a **tree** representation of the parsed input, because trees make the structure of the input obvious, and are easy to work with.

The file AST.java provides an interface AST that describes any Unicalc abstract-syntax-tree node: it has to have eval(...) and toString(...) methods. It also provides all the classes you will need for representing nodes in an abstract syntax tree for Unicalc, including:

- Product (of two subtrees)
- Quotient (of two subtrees)
- Sum (of two subtrees)
- Difference (of two subtrees)
- Power (of a subtree, to an integer power)
- Negation (a subtree) -- i.e., change the sign
- Normalize (a subtree)
- Value — an AST node that just contains a constant Quantity
- Define (a name in the database as the result of some sometree)

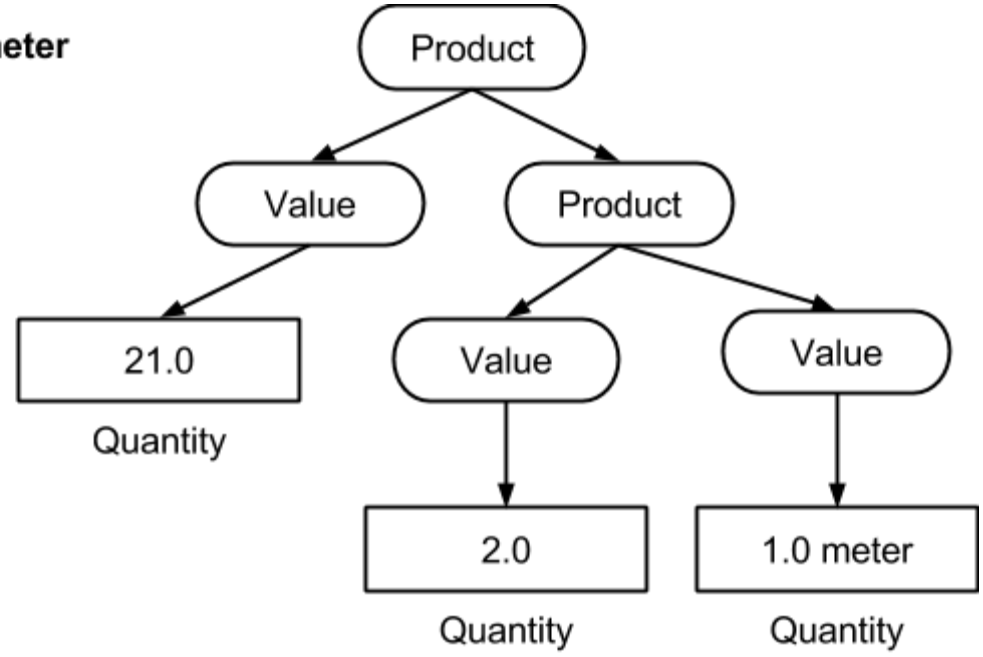
As provided, the eval(...) methods don't do much, but toString() and equals work, so those can be used to test the parser. In Part 3, you will add code to correctly evaluate the trees you are building here.

Here is the grammar again. The comments suggest how you might build and return abstract syntax trees, if the input matches the right-hand-side of the corresponding rule

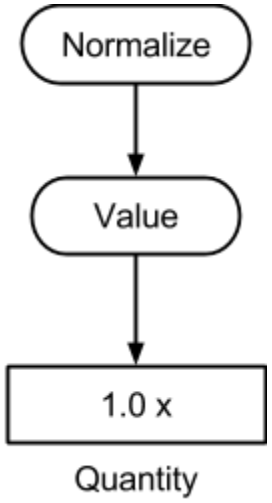
```
GRAMMAR              AST to return
----------           -------------
S -> def W L         new Define(..., ...)
   | L               whatever L() constructed

L -> # E             new Normalize(...)
   | E               whatever E() constructed

E -> P + E           new Sum(..., ...)
   | P - E           new Difference(..., ...)
   | P               whatever P() constructed

P -> K * P           new Product(..., ...)
   | K / P           new Quotient(..., ...)
   | K               whatever K() constructed

K -> - K             new Negation(...)
   | Q               whatever Q() constructed

Q -> R Q             new Product(..., ...)
   | R               whatever R() constructed

R -> V ^ J           new Power(..., ...)
   | V               whatever V() constructed

V -> D               new Value( new Quantity (...) )
   | W               new Value( new Quantity (...) )
   | ( L )           whatever L() constructed

J -> I | - I
I -> (any nonnegative integer)
D -> (any nonnegative number, integer or floating-point)
W -> (any word (that is a unit name))
```

Here are some more specific examples of possible inputs, and the corresponding Abstract Syntax Trees of objects that should be constructed.

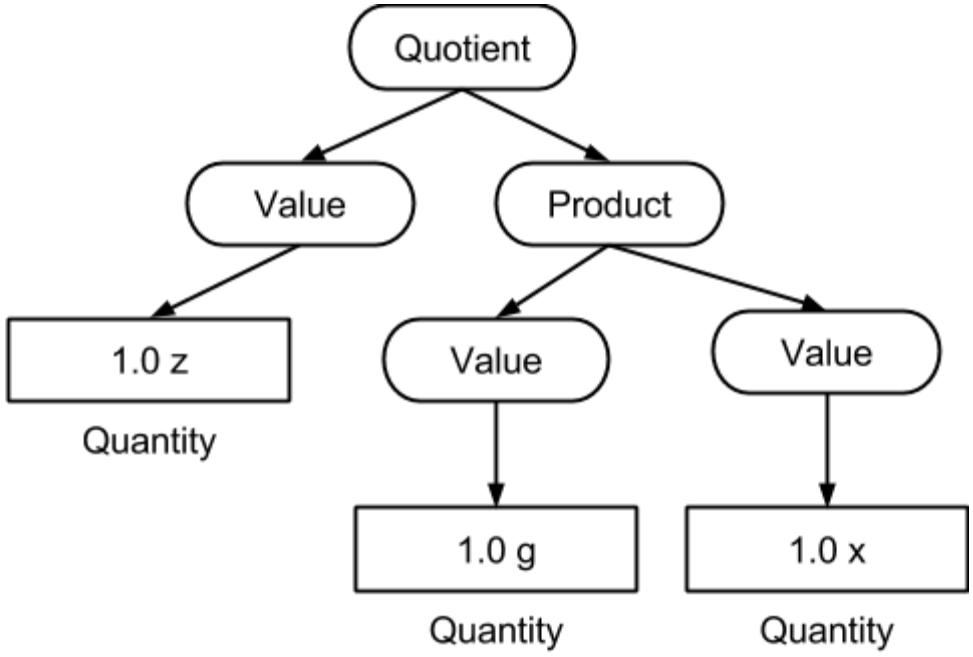
21 * 2 meter


Product
Value
Product
21.0
Quantity
Value
Value
2.0
Quantity
1.0 meter
Quantity

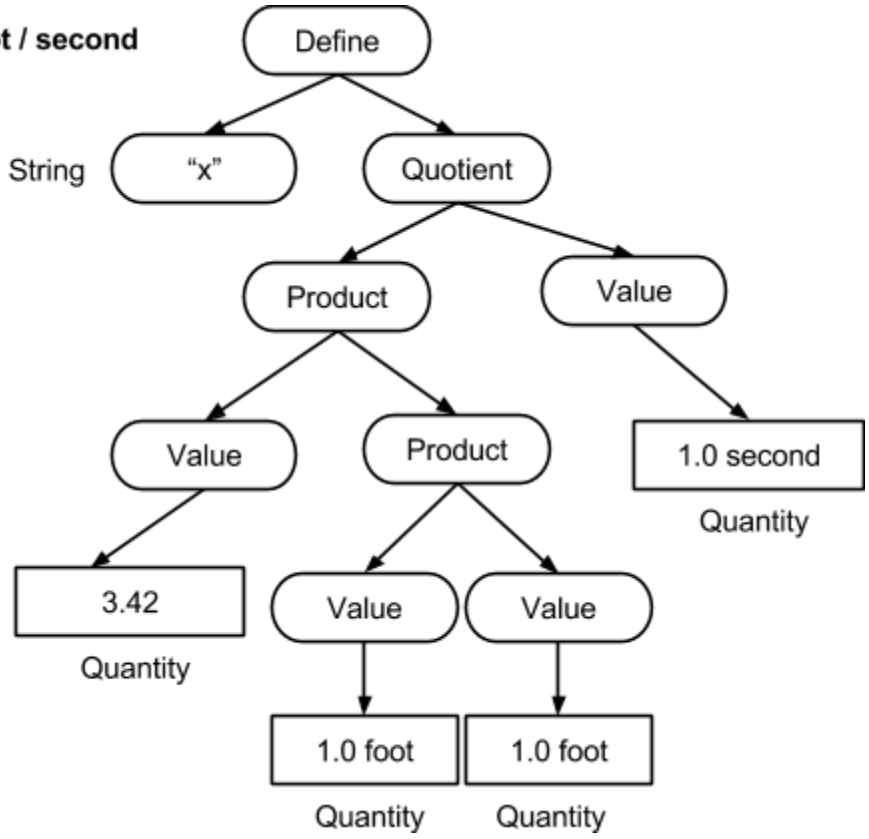
 # x


Normalize
Value
1.0 x
Quantity

z / g * x



def x 3.42 foot foot / second



Part 3: The Evaluator (AST.java)

This part completes the Unicalc language. Fill in code for the eval method in the AST classes (in AST.java). The code should work as follows:

- For most classe, eval should recursively evaluate the subtree(s), and perform the appropriate operation on the resulting quantities.
- Eval for Value objects should just return the quantity contained in that object.
- Eval for Sum and Difference should recursively evaluate subexpressions, normalize the results, and add/subtract quantities.
- Eval for Definition should recursively evaluate the subexpression, update the database appropriately, and return the result of the subexpression.

Grading

Points on this assignment will be awarded as follows:

- Style and documentation: 60 points
- Part 1 correctness: 140 points
- Part 2 correctness: 140 points
- Part 3 correctness: 60 points

Your checkpoint submission will be graded as follows:

- We will ensure that you have at least one test for each public method
- We will ensure that each of these tests are correct

If you do both of these things, you “pass” the checkpoint. If you do not pass the checkpoint, you will be penalized 10% of your final grade.

The goal of the checkpoint is to ensure that you have a good start on your assignment with at least a week to go.

Turning in your assignment (commands coming soon)

Checkpoint:

Final solution: