Arithmetic and Other Small Pebbles

Overview

In this machine problem you will start by implementing a calculator for common arithmetic operations. You will then generalize your work to perform simple algebraic manipulations and to compute the derivative of an arbitrary function using Newton's method. From here, you will use your algebraic framework to solve alphametic puzzles, also known as *cryptarithms*. Another aspect of this machine problem, to help you work with binary trees, is an exercise in debugging.

To help you plan your work, this MP has been split into three parts.

Part 1 concerns the calculator and the algebraic evaluation. **Part 2** concerns solving cryptarithms, and this part involves generating permutations and combinations as a sub-part. **Part 3** (although this is better completed early) is a debugging exercise involving a height-balanced binary tree called an **AVL Tree**.

The skeleton code provided is quite sophisticated. Reading source code is an important skill and you must read the source code first and create an illustration of the different components and how they link up. A formal approach to creating such a model of software is the Unified Modeling Language (UML) but there is no expectation that you use UML in CPEN 221.

Logistics

- Due Date: November 15, 2017 (11:59 p.m.)
- Course Grade Weight: 10%
- This assignment is to be completed in **pairs.**
- You may select your partner for this assignment.

Apart from making sure you understand how to work with `interfaces`, subtypes (and subclasses) and recursive types, you should also familiarize yourself with using command line arguments.

1. Calculators and Algebra

1.1. Common Operations

In `ca.ubc.ece.cpen221.mp4.operator` we have provided an Operator interface to represent an arithmetic operator and sub-interfaces to represent binary operators (such as addition, subtraction, multiplication, division, and exponentiation) and unary operators (such as negation and absolute value). To complete this part you must provide concrete implementations for these seven operators complete a short program that passes a list containing all of your operators to the constructor for a GUI calculator[1]. See the `main`[2] method in the GUI for details. When you are done you should be able to call the `launch` method on the resulting calculator and make sure your operators work; fix them if they don't. We recommend (but do not require) that you write additional operators to see how easy it is to extend the functionality of the calculator. Note that the calculator operators embody the **strategy pattern**; each operator represents a strategy for calculation.

[1] `ca.ubc.ece.cpen221.mp4.gui.Calculator`
[2] `ca.ubc.ece.cpen221.mp4.gui.Main.main`

1.2. Implementing Calculator Expressions

In `ca.ubc.ece.cpen221.mp4.expression`, we have provided an `Expression` interface to represent arithmetic expressions. In this part your task is to implement datatypes to represent numbers, unary operator expressions, and binary operator expressions. You can informally test your implementation by manually constructing familiar arithmetic expressions (such as `sqrt(3 * 3 + 4 * 4)`) and checking that they evaluate to the correct value. We have included a command line calculator to help you informally test your solution. Similar to the GUI calculator, the command line calculator takes a set of `Operators`; in the `main` method of `CommandLineParser`[3] you should add your operators to the operator set. The parser also needs to be able to construct the `Expressions` you write; implement the three functions in `ExpressionMaker`[4] by calling your respective `Expression` constructors, and then run the command line parser. The parser requires that spaces be placed between numbers and binary operators:

```
Enter an expression
1 + (2 * (3 - 4) / 5) - 2
Result: -1.4
Enter an expression
1+(2*(3-4)/5)-2
Input format not accepted. Please try again.
```

1.3. Functional Programming: Derivatives and Newton's method

You will first extend your work to support expressions containing named variables (e.g. `x*x`), and then use that solution to write an expression that numerically evaluates the derivative of another function. Finally, you will use your derivative expression in an implementation of Newton's method, which is a numerical algorithm to find the zeros of a function.

Expressions with named variables

`VariableExpression`[5] provides a skeletal implementation of an `Expression` representing a variable expression, essentially a named box to represent a value much like a variable represents a value in algebra. Complete that skeleton implementation and test it by creating a variable named `x` and an expression representing `x * x - 2`. You should not need any new constructors to do this; your `VariableExpression` and solution from earlier should suffice. Set `x` to some value and verify that the overall expression evaluates to the correct numerical value. If you would like, you can write a little program to generate a table of values for a function by repeatedly setting `x` and evaluating the function. Also you can (but are not required to) play with functions of multiples variables (e.g., `ax^2 + bx + c`).

An expression to compute the derivative

In calculus, the derivative of a function `f(x)` is another function `f'(x)` whose value at each point `x` is the slope of `f(x)` at `x`. The derivative of a function can be approximated with respect to a variable in that function as $f'(x) = (f(x+\delta x) - f(x))/\delta x$ where δx is some arbitrary small value. For this sub-part, write an implementation of the `Expression` interface whose instances represent the derivative of some specified function. The constructor for your implementation should resemble:

[3] `ca.ubc.ece.cpen221.mp4.parser.CommandLineParser.main`
[4] `ca.ubc.ece.cpen221.mp4.parser.ExpressionMaker`
[5] `ca.ubc.ece.cpen221.mp4.expression.VariableExpression`

```
/**
 * Creates an expression representing the derivative of
 * the specified function with respect to the
 * specified variable.
 *
 * @param fn
 *       the function whose derivative this expression represents
 * @param independentVar
 *       the variable with respect to which we're differentiating
 */
public DerivativeExpression(Expression fn,
                            VariableExpression indepedentVar) {
    ...
}
```

The `DerivativeExpression`'s `eval` method should return the approximate derivative of `fn` at whatever value `independentVar` is set, much like evaluating a function from the previous part. To approximate derivatives for this task, set `x` to be a private constant value `DELTA_X = 1e-9` (i.e., 10^{-9}). Test your implementation by evaluating the derivative of `x * x - 2` at various points; the result should be approximately equal to `2 * x`. Similarly, the derivative of `sin(x)` should be approximately equal to `cos(x)`.

Newton's method

Finally, use your `DerivativeExpression` to compute the zeros of a function using Newton's method. The zeros of a function `f(x)` are the values of `x` at which `f(x)` evaluates to zero. For example, the zeros of `x^2 -3x+2` are `1` and `2`.

Newton's method is a numerical algorithm that iteratively improves a coarse approximation of a zero until the approximation is sufficiently accurate. See the Newton's method Wikipedia article for details. Given an initial estimate `x_0` of a zero, Newton's method computes an improved estimate `x_1` as `x_1 = x_0 - f(x_0)f'(x_0)`. The same formula is used to compute each successive estimate: `x_(i+1) = x_i - f(x_i)f'(x_i)` and the process can be repeated until the estimate is sufficiently close to an actual zero. Newton's method fails for some functions `f(x)`, but is known to converge for many functions. To complete this task, write a method to compute a zero of an arbitrary function given an initial rough approximation and a target accuracy:

```
/**
 * Returns a zero of the specified function using
 * Newton's method with approxZero as the initial estimate.
 *
 * @param fn the function whose zero is to be found
 * @param x the independent variable of the function
 * @param approxZero initial approximation for the
 *        zero of the function
 * @param tolerance how close zero the returned
 *        zero has to be
 */
```

2. Cryptarithms

A cryptarithm (or alphametic) is a puzzle where you are given an equation with letters instead of digits. For example, one famous cryptarithm published by Henry Dudeney in 1924 is:

```
  SEND
+ MORE
------
 MONEY
```

To solve a cryptarithm you must figure out which digit each letter represents. Cryptarithms typically follow standard rules: The first letter of each word (in the above example, S and M) cannot represent zero, and each letter represents a different digit. Good cryptarithms have exactly one solution.

You can use logic to solve a cryptarithm by hand, but on a computer brute force works fine due to the small size of the search space. Because each letter represents a different digit there can be at most ten distinct letters, in which case there are only 10! (or 3,628,800) possible assignments of letters to digits. Because modern computers execute billions of instructions per second, checking 3.6 million possible solutions is easy.

2.1. A Reusable Permutation Generator

A permutation of a set of items is an arrangement of the items into an ordered sequence. One way to generate all possible assignments of k letters to digits (to solve a cryptarithm) is to generate all permutations of all size-k subsets of digits, assigning letters by simply matching the letters (in some fixed order) to the digits in each permutation.

To solve cryptarithms, (for this MP) we require that you design, implement, and use some form of permutation generator. Your permutation generator must be a reusable component that is not specific for — and does not depend on — cryptarithms or your cryptarithm solver. To build a general permutation generator you must make many design choices, including (but not limited to):

- The representation of a permutation.
- The represention of sets to be permuted, such as arrays, collections, `String`s, or other aggregate data types.
- Architecture-level design for the permutation generator and for how the generator provides access to the resulting permutations. We envision a wide range of possible solutions, possibly using design patterns such as the template method, iterator, and/or strategy patterns.
- The extent to which permutation generation is coupled (or decoupled) to subset generation for input sets.

These design problems are complicated by the fact that a permutation generator will typically be used as a component in a brute-force algorithm (such as the cryptarithm solver), and thus the permutation generator needs to be a high quality, high performance component. Performance requirements may constrain your architectural decisions but should not be the only non-functional requirements of your design. You may use any reasonable algorithm to generate permutations, but we recommend a standard technique such as Heap's Algorithm.

2.2. Representing Cryptarithms

Design and implement a datatype representing cryptarithms. Your Java implementation of this type should have a constructor that takes a single `String` array representing the cryptarithm. For example, the `String` array `{"SEND", "+", "MORE", "=", "MONEY"}` represents the cryptarithm above. Command-line arguments will be similarly passed to main if you run your program as `java <class name> SEND + MORE = MONEY`.

Note that mathematical equations can be represented as a pair of expressions, one expression for each side of the equation. You should use your `Expression` framework from **Part 1** to help you represent and evaluate cryptarithms. Notice that each letter in a cryptarithm is essentially a variable, and you can build an expression that represents (in terms of the letters) each side of a cryptarithm. This allows you to evaluate a potential solution by assigning a value to each variable, evaluating the expressions that form the cryptarithm's equation, and checking for equality.

For this task, a cryptarithm may use addition, subtraction, multiplication, and/or division. Each side of the cryptorithm may use an arbitrary number of operands and operators. Assume that all operands have equal precedence; this assumption greatly simplifies parsing, so you can parse a cryptarithm easily from left-to-right. If you are familiar with regular expressions or grammars, the following grammar might help you understand the cryptarithms your program must parse:

```
cryptarithm ::= <expr> "=" <expr>
       expr ::= <word> [<operator> <word>]*
       word ::= <alphabetic-character>+
   operator ::= "+" | "-" | "*" | "/"
```

In plain English, this grammar says:

- A cryptarithm consists of two expressions separated by an equals sign.
- Each expression is word optionally followed by one or more operator-word pairs. • A word is a sequence of one or more alphabetic characters.
- An operator is one of `+`, `-`, `*`, `/`.

Your cryptarithm constructor must throw an appropriate exception if the given sequence of Strings does not form a syntactically valid cryptarithm, or if the cryptarithm uses more than ten letters. (Recall that each letter in a cryptarithm must represent a distinct digit).

2.3. Solving Cryptarithms

Write a program that generates and prints all solutions to a cryptarithm. Your cryptarithm solver must use the permutation generator you wrote to generate possible assignments of letters to digits, and use your cryptarithm class and Expression framework from Part 2 to check whether each possible solution is valid. Also, a possible solution is not valid if the first letter of any word in the cryptarithm represents zero.

Your program should take a single cryptarithm as command-line arguments, with each token in the cryptarithm separated by one or more spaces. An example solution might look like this when you run it:

```
$ java SolveCryptarithm SEND + MORE = MONEY
  1 solution(s):
    {S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2}
$ java SolveCryptarithm WINTER + IS + WINDIER + SUMMER + IS = SUNNIER
```

```
  1 solution(s):
    {W=7, I=6, N=0, T=2, E=8, R=1, S=9, D=4, U=3, M=5}
$ java SolveCryptarithm NORTH / SOUTH = EAST / WEST
  1 solution(s):
    {N=5, O=1, R=3, T=0, H=4, S=6, U=9, E=7, A=2, W=8}
$ java SolveCryptarithm JEDER + LIEBT = BERLIN
  2 solution(s):
    {J=6, E=3, D=4, R=8, L=7, I=5, B=1, T=2, N=0}
    {J=4, E=3, D=6, R=8, L=9, I=5, B=1, T=2, N=0}
$ java SolveCryptarithm I + CANT + GET = NO + SATISFACTION
  0 solution(s)
```

3. Debugging AVL Tree

AVL trees are self-balancing binary search trees that you can learn more about here:

http://en.wikipedia.org/wiki/AVL_tree.

We have provided an AVL tree implementation, `AvlTreeSet.java`, in the package `avltree`[6], and its specification is included in the source code as Javadoc comments.

For this part of the assignment, you must:

1. Write unit tests to achieve 100% line coverage of the `AvlTreeSet` type.
2. Report an issue for each bug you find in `AvlTreeSet` on the GitHub issue tracker for your repository, which can be found at the following location:

   `https://github.com/CPEN-221/f17-mp4-<username>/issues`
3. Produce a high quality bug report each for each bug you find. Your report should include at least the following information: the context of the bug (i.e., where in the code the bug occurs), the error that occurs as a result of the bug, and (optionally) a suggested fix for the bug.
4. Fix each issue in an independent commit. Write meaningful commit messages and close the bug report. You can close an issue by adding 'fixes #xxx' to your commit message where #xxx refers to the issue number or by referring to the commit from the message with which you close the bug. You can read more about fixing issues with commits here. If you do not close the issue with a commit, please close the issue manually (on the GitHub issue tracker page) and refer to the commit on the issue page.

Note that this part of the assignment involves understanding how an AVL Tree works.

4. Some Hints and Advice

- When parsing words, use `Character.isAlphabetic` to check if a character is legal.
- Evaluating a cryptarithm is easier if you use a single `VariableExpression` to represent all repeated occurrences of each letter in the cryptarithm.
- There exist many good example cryptarithms on the web. We recommend Truman Collins's page and Torsten Sillke's page.

[6] `ca.ubc.ece.cpen221.mp4.avltree`

- Here is a trick that may help you generate all subsets of size `k` of a set of size `n`. Let each of the low order `n` bits of an `int` represent the presence or absence of an element:

```
for (int bitVec = 0; bitVec < 1 << n; bitVec++) { // 1 << n is 2^n
   if (Integer.bitCount(bitVec) == k) {
       /*
        * The positions of all of the one bits in bitVec represent
        * one combination of k int values chosen from 0 to n-1.
        */
  }
}
```

- The technique described above is not the most general or efficient, but it's good enough for this machine problem. If you choose to use it be sure you understand how it works. Read the documentation for `Integer.bitCount` if you don't know what it does.
- Use different Java files for different types of test cases. Clearly, tests for `AvlTreeSet` should be separate from tests for `Cryptarithm` but you can be more careful in your choices.

4.1. Learning Goals

- Understand and apply the concepts of subtyping, information hiding, and writing contracts, including an appropriate use of Java interfaces.
- Interpret, design, and implement software based on informal specifications, and demonstrate an understanding of basic software design principles.
- Use inheritance, delegation, and design patterns effectively to achieve design flexibility and code reuse.
- Write unit tests and automate builds and tests using JUnit, Gradle and Travis-CI.
- Understand the benefits and limitations of code coverage metrics and interpret the results of coverage metrics. Use tools such as Jacoco and Coveralls.io to measure test coverage.
- Demonstrate good Java coding and testing practices and style. Use tools such as Codacy to identify some problems with your coding practice.

4.2. Evaluation

A complete implementation will have the following characteristics:

- be submitted correctly and by the deadline to Github;
- pass all functional tests;
- include specifications, rep invariants and abstraction functions in the appropriate places;
- build successfully on Travis CI;
- include tests that achieve 100% code coverage (although this is not required for the GUI and the code that we have provided except `AvlTreeSet.java`) and have the results for code coverage appear on `coveralls.io`;
- have a project certification of A on Codacy;
- include (commit and push to Github) a file in the top-level directory called `work-division.md` (markdown formatting guidelines) that describes:
  - how the work for this machine problem was divided;
  - who did what;
  - hours spent on different aspects of the MP.

  - (Github records must show regular commits from both students in a team.)

4.3. Grade Breakdown

We will use the following weighting among the three *major* tasks.

- **Part 1**: 45%
- **Part 2**: 40%
- **Part 3**: 15%