Homework #3: From Cryptarithms to Algorithms

Due Sunday, February 7th at 11:59 p.m.

In this assignment, you will build a powerful cryptarithm solver using the expression evaluator framework you wrote for Homework 2. 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.

This is a three part assignment. In Part 1 of this assignment, you will design and implement a general purpose permutation and combination generator. In Part 2 you will use your expression framework from Homework 2 to write a class representing a cryptarithm. In Part 3 you will add a solve method to your cryptarithm, generating and checking all possible solutions using the permutation generator from Part 1.

Your learning goals for this assignment are to:

- Demonstrate mastery of earlier learning goals, especially the concepts of information hiding and polymorphism, software design based on informal specifications, and Java coding and testing practices and style.
- Use inheritance, delegation, and design patterns effectively to achieve design flexibility and code reuse.
- Discuss the relative advantages and disadvantages of alternative design choices.
- Gain experience working with a medium-sized application, including programming against existing interfaces and implementations.

Part 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 assignment 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.

When you are done, describe your design rationale in a short (at most 500 word) document, `rationale.md` (GitHub markdown) or `rationale.pdf` (PDF), in your `homework/3` directory. Explicitly discuss alternative designs that you considered, and highlight the design goals, design principles, and design patterns that guided your decisions.

Part 2: Representing cryptarithms

Design and implement a class whose instances represent cryptarithms. Your class 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 Homework 2 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 assignment 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 +, -, *, or /.

Your cryptarithm constructor must throw an appropriate exception if the given sequence of `String`s 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).

Part 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 in Part 1 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)
```

Testing your solution

You must formally test your solution using JUnit tests. When possible you should check the correctness of normal cases as well as edge cases of your implementation. We recommend that you start writing unit tests for your solution as you complete your solution; do not delay writing tests until after your implementation is complete. Remember that your permutation generator is an independent component that demands its own unit tests.

Evaluation

Overall this homework is worth 100 points. We will evaluate your solution approximately as follows:

- Mastery of earlier learning goals, especially the concepts of information hiding and polymorphism, software design based on informal specifications, and Java coding, specification, and testing practices and style: 60 points
- Use inheritance, delegation, and design patterns effectively to achieve design flexibility and code reuse: 30 points
- Discuss the relative advantages and disadvantages of alternative design choices: 10 points

Some hints and advice:

- Solutions to this assignment may vary. Please be sure that your solution develops and uses a general-purpose permutation generator and positively demonstrates the assignment's learning goals.
- 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.
- 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.
         */
    }
}
```

  This technique is not the most general or efficient, but it's good enough for this assignment. 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.