COS 126 Programming Exam 2

This exam is like a mini programming assignment. You will have 50 minutes to create two programs. Debug your code as needed. You may use your book, your notes, your code from programming assignments, the code on the COS 126 course website, the booksite, and you may read Piazza. No form of communication is permitted (e.g., talking, email, IM, texting, cell phones) during the exam.

Downloads. Before you begin (now), *download templates for your code and input data files,* as instructed for Programming Exam 2 on the COS126 Exams page.

Submissions. Submit your programs via the course website using the submit links as instructed for Programming Exam 2 on the COS126 Exams page. *Submit Part 1 before attempting Part 2.* You will lose a substantial number of points if you do not follow the submission instructions precisely.

Grading. Your program will be graded on correctness, clarity (including comments), design, and efficiency. You will lose a substantial number of points if your program does not compile or does not have the proper API, or if it crashes on typical inputs.

Discussing this exam. As you know, discussing the contents of this exam before solutions have been posted is a serious violation of the Honor Code.

Programming Exam: Finite Language Class

Part 1. Develop a class `Language` for processing finite formal languages, supporting union, concatenation, and *n*-closure.

Definitions. A *language* is a set of strings. A *set* is a collection of elements where no two elements are equal; we provide a class `SET.java` that enforces this property. The *union* of two languages is the set of all strings that are in either or both of the two languages. The *concatenation* of two languages is the set of all strings that can be formed by appending a string from the second language to a string from the first language. The *n-closure* of a language is the set of all strings that can be formed by concatenating *n* strings from the language. Here are some examples, using RE notation:

```
union:          a|abc = { a, abc }
concatenation:  (b|bc|bcd)(cd|d) = { bccd, bcd, bcdcd, bcdd, bd }
2-closure:      (e|ef){2} = { ee, eef, efe, efef }
```

Your task. Make sure that you have downloaded the files `Language.java` and `SET.java` as per the online instructions. `Language.java` is a client of the `SET` data type, which takes care of maintaining sets of *distinct* elements (adding a string to a set of strings that is already contains that string has no effect, as desired). Add code to `Language.java` as indicated within the file to implement the `concatenate()` method and the `closure()` method. We have provided implementations of the constructors and the `union()` and `toString()` method to help you get started.

Example. Note that `Language.java` has a `main()` client to test your methods by printing out the strings in the languages `a|abc`, `(b|bc|bcd)(cd|d)` and `(e|ef){2}`. Your program must behave as follows (the strings on each line can be in any order):

```
% java-introcs Language
 a abc
 bccd bcd bcdcd bcdd bd
 ee eef efe efef
```

Note that the string `bcd` appears only once in the second language, even though it can be formed either by concatenating `b` and `cd` or by concatenating `bc` and `d`.

Restrictions. Also as indicated within the file, you must use only a single instance variable `language` which is `final`. In other words, your `Language` class has to be *immutable*—the invoking `Language` object cannot change during a union, concatenate, or closure operation.

Hint 1. To implement these methods, you will need to use Java's for-each loop. See the provided `toString()` and `union()` methods for examples of using a for-each loop with `SET<String>`.

Hint 2. There are many methods in `SET.java` but the only ones you will need to use for this exam are: the constructor, the `add()` method, and for-each loops (see Hint 1).

SUBMIT `Language.java` AS INSTRUCTED ON THE COVER PAGE.

Part 2. Write a language client `PostfixL.java` with a single method `main()` that simulates a stack machine that takes commands from standard input to create languages.

Your task. Recall from Lecture 12 that the stack client `Postfix.java` simulates a stack machine that evaluates arithmetic expressions. Your program will operate in a very similar manner. Make sure that you have downloaded the file `Stack.java` as per the online instructions. Create an empty stack of `Language` objects, then use `StdIn.readString()` to read strings from standard input one at a time, until the standard input string is empty. For each string, respond as follows:

- If the string is `"UNION"` pop the top two languages and push their union.
- If the string is `"CONCATENATE"` pop the top two languages and push their concatenation.
- If the string is `"CLOSURE"` read an integer *n*, pop the top language and push its *n*-closure.
- If the string is `"PRINT"` pop the top language and print its strings.
- Otherwise, create a new language consisting of the string and push it onto the stack.

You may assume that the input is well-formed (you do not need to check for errors in the input).

Example. Suppose that the input is the file `testPostfixL.txt`:

```
a abc UNION PRINT
b bc bcd UNION UNION cd d UNION CONCATENATE PRINT
e ef UNION CLOSURE 2 PRINT
```

For this input, your program must behave as follows:

```
% java-introcs PostfixL < testPostfixL.txt
 a abc
 bccd bcd bcdcd bcdd bd
 ee eef efe efef
```

You may wish to test your program on your more complicated test cases, as we certainly will.

Note 1. Make sure that you use the `equals()` method in `String` rather than `==` to test whether two `String` values are the same sequence of character values.

Note 2. Make sure that you get the operands in the proper order for the concatenate operation. The second-to-top language is the first operand and the top language is the second operand.

SUBMIT `PostfixL.java` AS INSTRUCTED ON THE COVER PAGE.

Food for thought. Why not just use an array as the underlying data structure? The answer to this question is that the cost of *removing duplicates* would be prohibitive for large languages, and `SET` efficiently implements this important functionality. Also, with `SET`, we can provide clients with an efficient implementation of the operation of checking whether a given string is in the language.