CS2030 Programming Methodology
Semester 1, 2020/2021

21 October 2020
Problem Set #8 Lazy Evaluation

1. Let's explore the difference in computational efficiency between using `ArrayList` (which employs eager computation) and `LazyList` (which employs lazy computation). To do this, we will find the k^{th} prime number in a given interval of integers $[a, b]$ (eg. the 4^{th} prime number in $[2, 100]$ is 7) using the two different types of lists.

   Study the program listed in `Primes.java`; in particular, compare the methods `findKthPrimeLL` and `findKthPrimeArr`. Both use the same functional programming approach: (i) generate a list of integers in the given range, (ii) filter the list using the `isPrime` predicate, and (iii) retrieve the k^{th} element from the filtered list.

   The program is already written and compiled for you; you just need to run it. To do so:

   1. Read the Appendix for instructions on how to setup.
   2. In the `processed/` directory, run the program:

      ```
      java Primes 100000 200000 5 1
      ```

      This will find the 5^{th} prime in the interval $[100000, 200000]$ using `LazyList`. The program will report the prime number found, and also the number of times `isPrime` was called:

      ```
      The 5-th prime is 100057
      isPrime was called 58 times
      ```

   3. Type: `java Primes` to read further instructions on how to run the program.
   4. By choosing between the 2 methods of finding primes, compare how many times `isPrime` is invoked. Try large ranges to see the difference, eg. there are over 8000 primes in the interval $[100000, 200000]$.[1]

   Answer these questions:

   (a) To find the 5^{th} prime in the interval $[100000, 200000]$, why does `findKthPrimeLL` make so many fewer calls to `isPrime` compared to `findKthPrimeArr` ?

   (b) Does the number of calls depend on k, or on the interval $[a, b]$, or both?

   (c) Can `findKthPrimeLL` make more calls to `isPrimes` than `findKthPrimeArr`? Epxlain.

2. Two LazyLists may be *concatenated*, ie. a new LazyList is created whose elements are those in the first list, in order, followed by those in the second list, in order. Example:

   ```
   var s1 = LazyList.fromList(1, 2, 3);
   var s2 = LazyList.fromList(4, 5);
   s1.print();
   ```

[1] The Prime Counting Function, $\pi(x)$, is the number of primes up to and including integer x. Try it here: `https://www.dcode.fr/prime-number-pi-count`

```
=> (* 1, 2, 3, *)

s2.print();
=> (* 4, 5, *)

s1.concat(s2).print();
=> (* 1, 2, 3, 4, 5, *)
```

Note that `concat` will not work if the lists are infinite. Here's the code:

```
public LazyList<T> concat(LazyList<T> other) {
    if (this.isEmpty())
        return other;
    else
        return LLmake(this.head(),
                      this.tail().concat(other));
}
```

(a) Using `concat`, add an instance method `reverse()` to the LazyList class. Since LazyLists are immutable, `reverse()` should return a new list, whose elements are the elements of `this` list, but in reverse order.

(b) Try your code on: `LazyList.fromList(1, 2, 3, 4, 5).reverse().print()`

(c) Complete the code below to create a `flatmap` instance method. *Hint:* Use `concat`.

```
/** **************
    Apply the function f onto each element of this list, and
    return a new LazyList containing all the flattened mapped elements.
    Note that f produces a list for each element. But the returned
    list flattens them all, ie. removes nested lists.
    */
    <R> LazyList<R> flatmap(Function<T, LazyList<R>> f) {
        if (this.isEmpty())
            return LazyList.makeEmpty();
        else
            // insert code here
    }
```

(d) An $r-$Permutation of a list of n integers is a arrangement r integers, taken without repetition from the list. Example: $(3, 1, 2)$ is a 3-Permutation of $(1, 2, 3, 4)$; a different 3-Permutation is $(2, 3, 1)$. Here's how we may generate all the $r-$Permutations of a list L of n integers.

1. If $r = 1$, then this is a list of singleton lists of each of the elements in L. Example: for $L = (1, 2, 3, 4)$, there are four 1-Permutations: $((1), (2), (3), (4))$.
2. If $r > 1$, take each element x in turn from L, recursively compute the $(r - 1)$-Permutation of $L - x$ (ie. L with x removed). This will give a list of $(r - 1)-$Permutations. We now insert x to the front of each of these.

In the file `Puzzle.java`, complete the code implement the `permute` function.

```
LazyList<Integer> remove(LazyList<Integer> LL, int n) {
    return LL.filter(x-> x != n);
}

LazyList<LazyList<Integer>> permute(LazyList<Integer> LL, int r) {
    if (r == 1)
        return LL.map(x-> LLmake(x, LazyList.makeEmpty()));
    else
        // Insert code here
}
```

(e) Try it on: `permute(LazyList.intRange(1,5), 3).forEach(LazyList::print);` This should print ${}^4P_3 = 24$ 3-permutations. Note that the `forEach` instance method applies the given `Consumer` function onto each element of the list.

Read the Appendix to see how to compile your code.

3. A cryptarithmetic puzzle is shown below. Each letter represents a distinct numeric digit. The problem is to find what each letter represents so that the mathematical statement is true.

$$\begin{array}{cccc} & A & B \\ + & B & C \\ \hline A & X & Y \\ \hline \end{array}$$

In this example: $A = 1, B = 8, C = 4, X = 0, Y = 2$ is a solution to the puzzle. Other solutions are also possible, as you can easily determine.

(a) Let's try to solve the cryptarithmetic problem below by brute force, ie. checking all the possibilities. First, generate all the 6-Permutations of the list of 10 digits (0 to 9). These 6-Permutations correspond to our choice of C, H, M, P, U, Z. Next, check each 6-Permutation to see if it satisfies the given equation.

$$\begin{array}{ccccc} & P & Z & C & Z \\ \times & & & 1 & 5 \\ \hline M & U & C & H & Z \\ \hline \end{array}$$

Complete the definition of `pzczSatisfies` in Puzzle.java to solve the problem. To run it, increase the stack size of the Java Virtual Machine (JVM) to 8Mb as follows: `java -Xss8m Puzzle`. Otherwise, you may run out of stack space.

```
public class Puzzle {

    static boolean pzczSatisfies(LazyList<Integer> term) {
    // term is a list of 6 digits
    int c = term.get(0);
    int h = term.get(1);
    int m = term.get(2);
    int p = term.get(3);
    int u = term.get(4);
    int z = term.get(5);
```

```
        //Insert your code here.
        //Return true only when both m,p are not 0,
        //and the given equation is satisfied.
        }

        public static void main(String[] args) {
            permute(LazyList.intRange(1,10), 6)
                .filter(Puzzle::pzczSatisfies)
                .forEach(LazyList::print);
        }
    }
```

(b) Using a similar strategy, solve this problem:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Appendix: How to Compile

Code that use `LLmake` or `freeze` are non-standard Java. Special care is needed to compile it. This page explains how.

1. Use `bash` on Windows Subsystem for Linux (WSL), or MacOS, or Ubuntu, and `cd` to your `$HOME` directory: `cd ~`
2. Create two directories: `mkdir bin recit8`
3. Set permission: `chmod 755 bin; chmod 755 recit8`
4. Download `recit8-code.zip` from Luminus Files and copy it to `~/recit8/`, which will henceforth be called `$BASE_DIR`. On WSL, it is best to copy using `cp` instead of the Windows File Explorer (which may corrupt the file).
5. Unzip the file: `unzip rec8-code.zip`. You should see these files, and another directory called `processed`:

   ```
   ~/recit8=> ls -R
   .:
   LazyList.java  Primes.java  Puzzle.java  jpp  processed

   ./processed:
   LazyList.class  Primes.class
   ```

6. Make the `jpp` script executable: `chmod 755 jpp`
7. Move the file `jpp` to your `~/bin` directory: `mv jpp ~/bin/`
8. Add `bin` to your `PATH`: `export PATH=~/bin:$PATH`
9. Set your `CLASSPATH` to `$BASE_DIR/processed`.

   ```
   export CLASSPATH=~/recit8/processed:$CLASSPATH
   ```

10. The last 2 `export` commands can be made permanent if you write them into your `~/.bashrc` file. Otherwise you will have to `export` again the next time you start the bash shell.
11. Use the `jpp` script instead of `javac` to compile your `.java` files. This will translate `LLmake` and `freeze` (if any) into standard Java code, and then invoke the `javac` compiler for you. The `class` files will be placed in the `processed/` directory. For example:

    ```
    jpp Puzzle.java
    ```

    This will handle `LLmake` and `freeze`, **AND** invoke `javac` on `Puzzle.java`.
12. Always edit your `.java` files in `$BASE_DIR/` and not those in `processed/` (which will be overwritten by `jpp`). But if you wish to run `javac` with `-Xlint:unchecked`, then do this on the files in `processed/`.
13. Note that the 2 `class` files in `processed/` have been compiled, and are ready to run. Type: `java Primes` to run it.