Labs for Monday 20 May 2019

Last update 2019-05-17

Goal of the exercises

The goal of this week's exercises is to make sure that you can use Java 8 functional programming (including immutable data, functional interfaces, lambda expressions, and method reference expressions), and in particular Java 8 (parallel) streams and parallel array operations through mostly-functional programming.

Do this first

Get the lecture's example code from http://www.itu.dk/people/sestoft/itu/PCPP/E2016/pcpp-week03.zip and the Java Precisely 3rd edition example code from http://www.itu.dk/people/sestoft/javaprecisely/ and unpack both.

Exercise 1 In this exercise you will extend class FunList<T> of immutable singly-linked lists from Java Precisely Example154.java.

In the solutions, feel free to use recursion instead of explicit loops when convenient; do not worry about stack overflow here. In the descriptions below, the given list is assumed to have elements `x1`, `x2`, ..., `xn`.

1. Add a method `FunList<T> remove(T x)` that creates a new list in which all occurrences of `x` have been removed.

2. Add a method `int count(Predicate<T> p)` that returns the number of elements that satisfy predicate `p`. Remember to import the Predicate type.

3. Add a method `FunList<T> filter(Predicate<T> p)` that creates a new list that contains exactly those elements that satisfy predicate `p`.

4. Show how to implement an alternative version of `remove`, called `removeFun`, by a call to `filter` and using a lambda expression, and no explicit recursion or iteration.

5. Add a static method `FunList<T> flatten(FunList<FunList<T>> xss)` that creates a single list containing all the elements of the individual lists in `xss`, in order of appearance.

6. Show how to implement an alternative version of `flatten`, called `flattenFun`, using `reduce`, a lambda expression, and `append`, and no explicit recursion or iteration.

7. Add a method `<U> FunList<U> flatMap(Function<T,FunList<U>> f)` that computes the lists `f.apply(x1)`, `f.apply(x2)`, ..., `f.apply(xn)` and then returns the concatenation or flattening of these lists. Write both an explicit implementation, using recursion or iteration, and an implementation called `flatMapFun` using `map` and `flatten`, and no explicit recursion or iteration.

8. Add a method `FunList<T> scan(BinaryOperator<T> f)` that returns a list $y_1, y_2, \ldots, y_n$ of the same length as the given list $x_1, x_2, \ldots, x_n$. Here y_1 equals x_1, and for $i > 1$ it should hold that y_i equals `f.apply(`y_{i-1}`, `x_i`)`.

Exercise 2 In this exercise we will use parallel array operations to experimentally investigate the assertion that the count $\pi(n)$ of prime numbers less than or equal to n is proportional to $n/\ln(n)$, where $\ln(n)$ is the natural logarithm of n. More precisely, the ratio $\pi(n)/(n/\ln(n))$ converges to 1 for large n. This is known as the prime number theorem.

1. Create an `int` array `a` of size N, for instance for $N =$ 10,000,001. Use method `parallelSetAll` from utility class Arrays to initialize position `a[i]` to 1 if `i` is a prime number and to 0 otherwise. You may use method `isPrime` from the other prime number related examples.

2. Use method `parallelPrefix` from utility class Arrays to compute the prefix sums of array `a`. After that operation, the new value of `a[`i`]` should be the sum of the old values `a[0..`i`]`. Therefore, the new value of `a[`i`]` is the count of prime numbers smaller than or equal to `i`, that is, $\pi(i)$. For instance, the value of `a[10_000_000]` should be 664,579.

3. Use a for-loop to print the ratio between `a[i]` and $i/\ln(i)$ for 10 values of `i` equally spaced between $N/10$ and N.

Exercise 3 This exercise is about processing a large body of English words, using streams of strings. There is a list of some 235,000 English words and names in file /usr/share/dict/words on most Unix systems, including MacOS. If you use Windows or do not have the file for some other reason, get it from the course homepage at http://www.itu.dk/people/sestoft/itu/PCPP/E2015/words.zip

The exercises below should be solved without any explicit loops (or recursion) as far as possible.

1. Starting from the TestWordStream.java file, complete the `readWords` method and check that you can read the file as a stream and count the number of English words in it. For the `words` file on the course homepage the result should be 235,886.
2. Write a stream pipeline to print the first 100 words from the file.
3. Write a stream pipeline to find and print all words that have at least 22 letters.
4. Write a stream pipeline to find and print some word that has at least 22 letters.
5. Write a method `boolean isPalindrome(String s)` that tests whether a word `s` is a palindrome: a word that is the same spelled forward and backward. Write a stream pipeline to find all palindromes and print them.
6. Make a parallel version of the palindrome-printing stream pipeline. It is possible to observe whether it is faster or slower than the sequential one?
7. Write a stream pipeline that turns the stream of words into a stream of their lengths, then finds and prints the minimal, maximal and average word lengths.
8. Write a stream pipeline, using method `collect` and a `groupingBy` collector from class Collectors, to group the words by length. That is, put all 1-letter words in one group, all 2-letter words in another group, and so on, and print the groups. Optional challenge, easily answered by *Java Precisely*: Use another overload of `groupingBy` to compute (and then print) the number of 1-letter words, the number of 2-letter words, and so on.
9. Write a method `Map<Character,Integer> letters(String s)` that returns a tree map indicating how many times each letter is used in the word `s`. Convert all letters to lower case. For instance, if `s` is the word “Persistent” then the tree map will be `{e=2, i=1, n=1, p=1, r=1, s=2, t=2}`.

   Now write a stream pipeline that transforms all the English words into the corresponding tree map of letter counts, and print this for the first 100 words.
10. Use the tree map stream and the `reduce` method to count the total number of times the letter `e` is used in the English words. For the `words` file on the course homepage the result should be 235,886.
11. Words `s1` and `s2` that have the same tree map of letter counts (by `letters(s1).equals(letters(s2))`) are anagrams: they use the same letters the same number of times. For instance, “persistent” and “prettiness” are anagrams; both have letter counts `{e=2, i=1, n=1, p=1, r=1, s=2, t=2}`. Use the `collect` method on the word stream, `groupingBy` collector and the `letters` method to find and print all sets of anagrams in the file of English words. This may take 15–30 seconds to compute.

    For the `words` file on the course homepage the result should be 15,287 sets of anagrams, including `{a=1, c=1, e=1, r=1}=[Acer, acre, care, crea, race]`.
12. Try to make a parallel version of the anagram-printing stream pipeline. Is it faster or slower than the sequential one? (If your computer and Java version behaves like mine, this example shows that just slapping on `.parallel()` does not necessarily lead to more efficient execution).

Exercise 4 This exercise concerns various ways to create streams of doubles and of computing the sum

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}$$

for a large number N. For $N = 999{,}999{,}999$, the exact result should be near 21.300481501347944016685l.

1. Create a finite DoubleStream `1.0/1, 1.0/2, 1.0/3, ..., 1.0/N` where N = `999_999_999`, maybe using `IntStream.range` and `mapToDouble`. Compute the sum of such a list and print the result with a suitably large number of digits, for instance using

   ```
   System.out.printf("Sum = %20.16f%n", sum)
   ```

2. Now compute the sum also in parallel by inserting `.parallel()` in a suitable place in your stream pipeline. Print the computed result. Measure the time to compute the sum sequentially and in parallel.

3. Write a classic sequential for-loop to compute the sum, and print the result. Measure the wall-clock time used.

   Mysteriously, the result produced by your for-loop may be less accurate than that produced by the `sum` method on streams. This has little to do with streams, and more to do with how computers represent and compute with floating-point numbers of type `float` and `double`. Unlike real mathematical addition, computer floating-point addition is not associative, so we may have `(x + y) + z != x + (y + z)`. What the result really shows is that `sum` performs the summation in a more clever way than your for-loop! Presumably, method `sum` uses Kahan summation. This cleverness is necessary for parallelism; without it, `sum`'s result on a parallel stream of floating-point numbers might vary considerably from run to run due to the different orders in which the floating-point numbers are added: because computer floating-point addition is not associative, the order of summation matters.

4. Create the same finite DoubleStream as above, now imperatively using methods `generate` and `limit` on class DoubleStream. Compute the sum of the stream sequentially and print the result. Measure the time consumption.

5. Now try to add `parallel()` between `limit()` and `sum()` in an attempt to compute the sum in parallel. What happens with the result? What happens if you run the program multiple times? What does this say about the wisdom of using mutable state (in `generate`) in connection with parallel streams?