OPL Final Exam

Date: Thursday, April 2nd, 2020
Instructor: Rachata Ausavarungnirun

Problem 1 (20 Points):	
Problem 2 (20 Points):	
Problem 3 (20 Points):	
Problem 4 (20 Points):	
Problem 5 (20 Points):	
Extra Credit (10 points):	
Total (100 points):	

Instructions:

1. This is a 48-hour exam.
2. Submit your work as a zip file on Canvas.
3. Because this is a 48-hour exam, I expect everyone to `test your code`. Hence, you must **submit the test cases for all the coding questions** along with the code and explain what each test case is used for. Test cases, along with the explanation of how you test your code, will factor into 10% of all the questions.
4. If not specified, input and output types are a part of the question. Please use appropriate input and output types that make sense for the purpose of the question.
5. Please clearly comment your code, especially if your code do not work perfectly,
6. Clearly indicate your final answer for each conceptual problem.
7. Our typical restriction for Scala and Rust packages are similar to Assignments 3 and 4.
8. **DO NOT CHEAT.** If we catch you cheating in any shape or form, you will be penalized based on **my plagiarism policy** ($N*$ 10% of your total grade, where N is the number of times you plagiarized previously).

Tips:

- **Read everything.** Read all the questions on all pages first and formulate a plan.
- **Be cognizant of time.** The submission site will close at 00.01AM on Saturday.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.

1. ReadAloud ... Again [20 points]

In this question, we refer to the readAloud problem from Assignment 3. However, we are going to embrace our differences: each one of us will read this list differently.

In Scala, implement three different versions of *readAloud* using dependency injection. The three versions should behave the following way.

1. The first version of *readAloud* is called *mirrorReadAloud*, and will read the list the same way as the input list (seriously, this part should be free points).
2. The first version of *readAloud* is called *countingReadAloud*, and will read the list the same way as we define in assignment 3 (basically grouping the repeated elements together.
3. The first version of *readAloud* is called *unorderedReadAloud*, and will read each individual items in the list in any order. For example, if the list is [1, 2, 3, 4, 4, 5], *unorderedReadAloud* can return [1, 5, 2, 4, 3, 4].
4. The first version of *readAloud* is called *stutteredReadAloud*, and will read each element in an order similar to *mirrorReadAloud*. However, if the current element does not have any repeated value next to it, *readAloud* will read this element twice. For example, if the list is [1, 1, 1, 2, 3, 4, 4, 5], *stutteredReadAloud* will return [1, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5]. Note that 2, 3, 5 got read twice, but 1 and 4 do not because there is more than one copy of 1s and 4s.

Please save the file with the name `readAloudAgain.scala`

2. QuickSort in Scala [20 points]

In class, we show you how to implement a quicksort in parallel in Scala. Use the same algorithm and Future, implement a function *quickSort.* This function must take in any list that its elements have an `Ordered` trait (See `https://www.scala-lang.org/api/2.5.1/scala/Ordered.html` for more information), and produce a sorted list.

Please save the file with the name `quickSort.scala`

3. Huffman Encoding Redux [20 points]

In this question, we come back to our old friend from the midterm exam: the Huffman Encoding. Feels free to refer to the definition from the midterm exam. This question focuses on whether we can parallelize two of the sub-questions you implemented in Rust.

(a) **Parsing the data** [10 points]

For the first question, we will focus on the function called *CharCount*. *CharCount* takes in a string, and return a list of pair, the first element of the pair represent the character including a space and the second element represent the number of times that character is used in the input string.

Write a parallelize version of this function in Rust. The function should be named *CharCount*, takes in a string as an input, and produce a **vector** of tuple that consist of a character and an integer representing how many time the character appear on the input string. *Show the word and the depth of your implementation.*

(b) **InsertPQ** [10 points]

The next task has to do with insertion into our priority queue. In this task, implement a function *InsertPQ* that **takes in one element and one *sorted* vector**, and then **return another *sorted* vector** with the one additional input element inside. Basically, this function should insert this input element at the correct location in the vector.

To parallelize this process, you can find the correct location to be inserted in parallel. However, you can sequentially write the resulting vector serially (while you can also write the results into the new vector in parallel, I do not expect you to be able to deal with parallel mutable vector). *Show the word and the depth of your implementation.*

Please save your code using the filename `huffman.rs`. For the work and the depth, you can use the comment on the code itself to explain, or submit a separate file called `huffman.pdf` to show your work.

4. List and Newton's Method in Rust [20 points]

This question ask you on two unrelated topics.

(a) **Linked List is Bad [5 points]**

Explain why a linked list is not very good when using with Rust? What are the example of list operations that can become a performance bottleneck or erroneous results?

Please save the file with the name `linkedlist.pdf`

(b) **Newton's Method for Sqrt [15 points]**

In class, we went over the concept behind the Newton's method to approximate the value of square root. In this problem, you must come up with an implementation of a square root function called *sqrt* using the newtonian method, and calculate the work and depth of your implementation.

Please save the file with the name `sqrt.rs`

5. Optimizing Ackermann [20 points]

In this question, you will optimize the Ackermann function in Rust against your friends. Please call your function *ackermann*, which is defined as

```
ackermann(m,n) = n+1                            if m = 0
       = ackermann(m-1,1)                       if m > 0 and n = 0
       = ackermann(m-1, ackermann(m, n-1)) if m > 0 and n > 0
```

Your grade to this question will depend on how fast your code perform against your friends, both with low and high value inputs. The fastest 30% of the submission will get full credit. Then, your grade will goes down at the increment of 5%. An incorrect implementation will not get any score (so, please check for correctness).

Please save the file with the name `ackermann.rs`

Extra Credit: Basic Optimizing Compiler [10 points]

In this question, we will try to optimize a program based on the compiler techniques we learned from the last section of this class.

First, draw the dataflow diagram of basic blocks (with the code inside each basic block of the following code

```
i = a+b;
j = a-b;
c = i+j
if(i<j)
{
    i = b+a;
    a = i+j;
}
else
{
    j = a-b;
    b = i+j;
}
c = i+j+a+b;
```

Then, please rewrite the code above after we apply one pass from the following optimizations: value numbering → copy propagation → constant propagation → common subexpression elimination → dead code elimination.

After you finish with your optimization, redraw the dataflow diagram showing the new basic block along with the live variables entering and exiting each basic blocks.

Submit your work using the file name `extra-credit.pdf`