Programming Languages (CS 421), Spring 2019
Assignment 02 (85 points)
Due 5:00 PM, Friday, 08 March 2019

This assignment involves programming in Scala. You will write all of your code within the `main()` function of the supplied `HW02.scala` file; each part will involve defining a function and then calling it on some provided inputs. As you go, you are supplied with lines of testing code that you will un-comment to display results of your work; you can use this code to test your own, as each line contains a commented notation of the results you should see. **Leave this code un-commented** in your final submission, so that I can see the work on display.

For each problem, read the specification of what you are to do carefully. Many of the problems involve solving the problem in one or two different ways, or using particular techniques. Full points will be given for code that solves the problem in the correct manner, producing the correct results.

When you are done, submit the single source-file on D2L.

[**1**] (*5 pts.*) Implement two functions that take a pair of integer inputs, x and y, and return x^y. This must be achieved using only elementary Scala control structures and basic integer arithmetic (i.e., do not use any `math` library functions that already compute the result for you).

- Your first version, called `power`, should work **iteratively**. That is, it should define some `var` and loop to calculate the result. Proper Scala style means that no `return` statement appears.
- Your second version, called `powerRec`, should work **recursively** and functionally. There must be no use of `var` and no looping.

[**2**] (*5 pts.*) Implement two functions that take two integers inputs x and y and compute the sum of all values between the two, inclusive. (If $x = y$, this result will be x.) To do so, you may want to import `scala.math._` to use its `min` or `math` functions. Other than that, you can use elementary arithmetic.

- Your first version, called `rangeSum`, should work **iteratively**, using some `var` and loop.
- Your second version, called `rangeSumRec`, should work **recursively**. That is, there must be no use of `var` and no looping.

[**3**] (*10 pts.*) Implement two functions that take a list of integers as input, and return the product of all elements in the list. An empty list should return 0. In both cases, you should not use anything beyond basic Scala control structures and elementary arithmetic operations.

- Your first version, called `product`, should work **iteratively**. That is, it should define some `var` and loop to calculate the result.
- Your second version, called `productRec`, should work **recursively**. That is, there must be no use of `var` and no looping. A `match` expression is an appropriate vehicle for this function.

[**4**] (*10 pts.*) Implement two functions that take an integer x and a list of integers as input, and return `true` just in case $x \geq e$ for **every** element e in the list. If the list is empty, this will always be `true`. In both cases, you should not use anything beyond basic Scala control structures and elementary comparison operations.

- Your first version, called `geqList`, should work **iteratively**. That is, it should define some `var` and loop to calculate the result.
- Your second version, called `geqListRec`, should work **recursively**. That is, there must be no use of `var` and no looping.

[**5**] (*5 pts.*) In a functional language like Scala, functions can serve as inputs to other functions. The code you are given features a simple example, where the `apply` function takes an input parameter `fun` that is a function from integers to integers. The `plus1` function is then input to `apply`. Your code should take your last function, the recursive version that checks whether or not x is greater than every element of the list, and modify it so that now it takes in 3 parameters: a list of integers, the integer to compare, and any function f that takes in two integers and returns a `Boolean`. Your new version should then check if $f(x, y)$ is `true` for input x and every element y of the input list. You should call this one `functionListRec`.

[**6**] (*10 pts.*) Write a function called `countIn` that takes a list of `Any` type, and an object x of `Any` type, and returns the number of times that object occurs in the list. You should use a recursive solution, without any `var` elements, and a `match` statement to process the list. Your solution can use basic Scala control structures and operators, but no library functions; you can use the basic list concatenation operator (`::`).

[**7**] (*10 pts.*) Write a function called `pairsum` that takes a list of integer pairs (x, y) and returns the list of integers formed by summing each pair $(x + y)$. You should use a recursive solution, without any `var` elements, and a `match` statement to process the list. Your solution can use basic Scala control structures and operators, but no library functions; you can use the basic list concatenation operator (`::`).

[**8**] (*10 pts.*) Write a function called `pairdel` that takes a list of pairs (e_1, e_2), where each element can have any type at all, and an element x, which may also have any type, and returns the list of pairs formed by deleting all pairs from the original such that either $x == e_1$ or $x == e_2$. You should use a recursive solution, without any `var` elements, and a `match` statement to process the list. Your solution can use basic Scala control structures and operators, but no library functions; you can use the basic list concatenation operator (`::`).

[**9**] (*10 pts.*) Write a function called `sublist` that takes a list of any type, and two indices `start` and `end`, and returns the sub-list of elements from positions `start` to (`end` $- 1$). If the list is empty, or `start` $\geq$ `end`, then the empty list should be returned. If the `end` index is greater than the length of the list, then everything from `start` to the end of the list should be returned. You should use a recursive solution, without any `var` elements, and a `match` statement to process the list. Your solution can use basic Scala control structures and operators, but no library functions. You should not use element indexing to access any elements of the list directly; you can use the basic list concatenation operator (`::`).

[**10**] (*10 pts.*) Write three functions. Each will take in two lists, along with a function on pairs of elements, and return the list that results from applying the function to the first element in each list, followed by the result of applying the function to the next two elements, and so forth. If one list is longer than the other, it should return the result for the first n pairs of elements, where n is the length of the shortest of the two lists.

- The first version, `applyIntLists`, should take two lists of integers, and a function from pairs of integers to integers.
- The second version, `applyLists`, should take a generic type argument, and ensure that the lists, as well as the function, all use the same specified type. The sample code shows you a function, `prln[T]()`, to demonstrate Scala generic syntax.
- The third version, `applyLists2`, should take **three** generic type arguments: the type of the first list, the type of the second list, and the type of the output list. Modify the code to create such a method, which will need to take a function of the proper type. The code features a sample function, `prln2()`, for guidance.