Exercise 4 - Higher-order programming

Task 1:
a)
A = 2, B = 1 and C = -1 gives the following:
RealSol = true
X1 = 0.5
X2 = -1

A = 2, B = 1 and C = 2 gives the following:
RealSol = false
X1 = nil
X2 = nil

b)
Procedural abstractions are useful because

1. It allows for better reusability - you don't have to replicate the code several places.
2. It can be used to "hide" code, and thus be a way of managing problem complexity - the programmer can focus on the essential details and "ignore" the lower level details.

c)
Functions are syntactic variants (or linguistic abstractions) of procedures, and they behave in the same way except they always return a value.

Task 3:
d)
For Sum and Length, using left fold or right fold is equivalent because they both use + (plus) as an operand, and a + b will always be the same as b + a. It would be different for subtraction, as a - b and b - a are not the same if a does not equal b.

e)
For the product of list elements, a good value for U would be 1. Multiplying anything by 1 just returns the same value. 0 would not be a good value, because it would return 0 as the product every time.

Task 5:
b)
LazyNumberGenerator takes in a start value and appends a function that returns LazyNumberGenerator again but with an incremented start value. This recursion allows us to simulate an infinite list that is dependent on what we require from it. What this means is that if we only want the first value we will get a list where the head is our start value and the rest is a function that returns a list with a start value + 1. So each 'next' value we want from the infinite list represents a call to the function itself. The structure is build 'on demand' instead of all at once (lazy).

The limitation to this is mentioned above, namely that the function merely simulates an infinite list. Since we're not actually returning a complete list, but a list that requires a function call, we are not able to do actions on it as we would a regular list, such as {LazyNumberGenerator 0}.2.2.1.

Task 6:
b)
Benefits of the tail recursion approach:

1. There is no need to retain a stack frame, resulting in less burden on the system (especially with a deep stack!)
2. Better readability. It is easier to think iteratively than recursively.

c)
The programming language itself is not really relevant when it comes to tail recursion optimization, as whether or not it is accounted for is totally dependent on the compiler. One compiler for one language may do it, while another compiler for the same language may not.

One reason not to do tail recursion optimization is because debug builds often run without optimizations to get stack traces and other debug functionalities.