Task 1:

b) Procedural abstractions are useful because

1. You don't have to replicate code in several places
2. Users can ignore "how the code works". If you give the function a good name, the user will know how to use it and what to expect it to do.

c) The difference is that a function must return a value, while procedures is just a set of commands which can be executed in order. We can think of functions like a mathematical function, I.e f(x) = x^2. It will return x^2 based on the input.

Task 3:

d) For sum and Length, it does not matter if we use Right or Leftfold. The result will be the same. I.E {RightFold [1 2 3 4] fun{$ X Y} X + Y end 0} will give us 0 + 1 + 2 + 3 + 4 = 10 and {LeftFold [1 2 3 4] fun{$ X Y} X + Y end 0} will give us 0 + 4 + 3 + 2 + 1 = 10. Length are only counting the elements, adding one each time it encounters an element, so that will give us the same with both RightFold and LeftFold. Subtraction however will give a different answer for the two. I.E (I'm using the first element doubled as the initial value) {RightFold [1 2 3 4] fun{$ X Y} X – Y end 2} will give us 2 – 1 – 2 – 3 – 4 = -8, while {LeftFold [1 2 3 4] fun{$ X Y} X -Y end 8} gives us 8 – 4 – 3 - 2 - 1 = -2.

e) A good value for U in multiplication is 1. 0 is not a good value, because the initial value is what we are multiplying with the first time. So if we multiply with 0, the whole expression will equal 0. Multiplying with 1 gives us the correct answer.

Task 5:

b) My LazyNumberGenerator function returns a list containing the StartValue and an anonymous function that calls the LazyNumberGenerator with StartValue + 1. This function is triggered when we call {LazyNumberGenerator 0}.2 (when we access the second element of the list). This function will call LazyNumberGenerator and return the list [StartValue+1 anonymous function]. If we access the second element again, the function will return [StartValue+2 anonymous function] and this goes on until we access the first element which is StartValue + n, where n is the number of times we have accessed the second element of the list. The limitations I find is that if we want N numbers, we must call the function N times, we cannot get 2 or more numbers from one call. The function does not recall previous numbers.

Task 6:

b) When using tail recursion our stack is of a constant size. We only need to save the result (which is computed after each recursive call) and not every element of the list to sum at the last recursive call. Therefore we are using the memory in a more efficient way.

c) No, not all programming languages benefit from tail recursion. Some programming language's compiler handle optimization for tail recursive calls, while some, like python, does not. In I.e python's case, tail recursion will use more system resources than not using tail recursion.