Assignment #1 – Part 1: Theoretical Questions

*אני עולה חדשה ולכן רושמת חלק זה באנגלית

1. We will explain the following programming paradigms:
   (a) Imperative: Imperative programming is a programming paradigm that uses commands in which alter the state of a program. We define a commands as a logical unit for executing a program. So, in other words, an imperative program is made up of orders for the machine to execute. The aim of imperative programming is to describe how a program works. For instance, we can think of these statements as steps in which modify the state of the computer. In simple words, instead of describing what the program should do, imperative programming tells the computer exactly how to do it.
   (b) Procedural programming is an imperative programming style (a programming paradigm derived from imperative programming) in which a program is composed of one or more procedures (also called functions or subroutines). Procedures are purely a set of computational commands that must be completed, in which define a clearly specified interface of input and output parameters. This type of programming paradigm uses a top-down structured program approach and relies on planned actions. The procedures interface consists of a name, input parameters, and a return value, while using local variables inside the procedure and therefore not affecting the program state outside the scope of the procedure. In conclusion, this programming paradigm allows for the division of a large program to be broken down into smaller and more manageable pieces called procedures which are in accordance with a specified behavior.
   (c) Functional programming is a programing paradigm in which involves applying and writing functions to create programs. It is a declarative programming paradigm (focuses on results instead of the process) in which function definitions are expression-driven and map between values instead of a series of imperative statements. An expression is assessed to generate a value, while a statement is executed to assign variables for example. This programming model advocates the use of pure functions (no mutual state or mutation), in addition to referential transparency (the value of an expression relies solely on its sub-expressions). The use of pure functions allows for concurrent desires, eliminating the need for dangerous locking mechanisms. In this programming paradigm we want to avoid mutation and therefore allow only the use of const variables as well as no loops or any concepts that mutate something in which was declared, as we want only function-driven based programming.

   - The procedural paradigm improves over the imperative programming paradigm by several features. Even so that in imperative we have a more succinct program describing what we want to accomplish, we isolated parameters on which we apply the task on from the task itself. Therefore, we still must copy the code for each run on various parameters. Furthermore, there is merely accidental binding between the variable parameters and the actual code, and therefore this does not tell us anything about the relationship between the code and the variables themselves, causing irrelevance between these variables and all other code. Now in procedural programming, by

defining a procedure interface with a well-defined structure and behavior(such as using input/output parameters and a name) and using local variable, we overcome these weaknesses. It's significant that the procedure has a name: it lets us replace a sequence of instructions, and programmers may reuse the current procedure by simply remembering its name. Procedures have parameters (in the case above, the numbers parameter) and local variables which make this easier. This reduces code duplication.

- The functional paradigm improves over the procedural paradigm in a number of ways. First, we will emphasize some issues of procedural programming. Procedural programming facilitates shared states in combination with mutation, making concurrency problematic. Also, Procedural programming sticks to a step-by-step approach to implementing operations from the start, preventing efficiency improvements. This programming paradigm also makes it hard to construct extremely usable function abstractions, while making it harder to easily understand the outcome of the code because of shared states and alterations. Functional programming addresses some of these issues by encouraging computation which is immutable. In addition, it prefers to use more abstract operations (such as map, etc.) making code more reusable. We can also pass functions as parameters and return functions as computed values in a very easy way using this paradigm, facilitating lots of abstraction means and achieving functional abstractions.

2. We will write the most specific types for the following expressions:
   (a) (x, y) => x.some(y)
   <T> (x: T[], y: (t: T) => boolean) => boolean
   (b) x => x.reduce((acc, cur) => acc + cur, 0)
   (x: number[]) => number
   [*(x: number[]) => x.reduce((acc: number, cur: number) => acc + cur: number, 0: number)*]
   (c) (x, y) => x ? y[0] : y[1]
   <T> (x: boolean, y: T[]) => T

3. The concept of “abstraction barriers” relies on data abstraction. The basic concept behind data abstraction is to define a basic set of operations to each type of data object, such that all manipulations of that type of data can be represented, and then to manipulate the data using only those operations. The form of a certain programming scheme can be visualized as an abstract hierarchy tree, where the lines in which separate each level of the tree reflect abstraction barriers that separate the system's various layers. The barrier distinguishes the programs that use the data abstraction (above) from the programs that implement the data abstraction (below) at each step. For instance, in the image below, the horizontal lines represent abstraction barriers.