Part 1 – Theoretical questions

1. 1. פרדיגמה אימפרטיבית –

פרדיגמה שתבנית התכנית היא רצף מפורש של פקודות, כך שהרצת התכנית מתבצעת ביצוע הפקודות זו אחר זו. כל פקודה מבצעת שינוי כלשהו במערכת (לדוג' Java, C++, Python מממשות את הפרדיגמה).

* 1. פרדיגמה פרוצדוראליות –

\* על פי הפרדיגמה התוכנית מחולקת לפרוצדורות, כלומר לרצפים נפרדים של פקודות ("קטעי קוד" נפרדים). הפרדיגמה מפרקת את תכנית המקור לתתי תוכניות קטנות מה שמאפשר תחזוקה קלה יותר. בנוסף על כן היא מאפשרת לייצר היררכיות ופרוצדורות מושרשות (מקוננות).

* 1. פרדיגמה פונקציונאליות –הפרדיגמה מיועדת לחישוב ביטויים. תכנית בפרדיגמה זו היא ביטוי/סדרת ביטויים ולא רצף של פקודות, כך שגם פונקציות הן ערכים. מציאת הערך של כל ביטוי מתבצעת בעת הרצת התכנית. בשפות פונקציונאליות אין פעולת השמה. מאחר ופונקציות גם נחשבות כערך, ניתן להעבירן כפרמטר ואף להחזיר פונק' כפלט מפרוצדורה פונקציונאלית וכך לבנות תכנית על ידי קריאות משורשות ורקורסיביות של פונקציות.
* הפרדיגמה הפרוצדוראלית משפרת את הפרדיגמה האיפרטיבית בכך שהיא מאפשרת לתחום בפרודצדורה רצף פקודות שמשומשות מספר פעמים לאורך הקוד (ובכך למנוע חזרתיות). בנוסף, על ידי תחימת הקוד בפרוצדורה שמאפשרת קבלה של קלטים, אנו מאפשרים התאמה של הקוד למקבץ של ערכים אפשריים. לעומת זאת, בתכנות אימפרטיביות, התאמה לערכים אחרים דורשת שינוי של הקוד.
* הפרדיגמה הפונקציונאלית משפרת את הפרדיגמה הפרוצדוראלית באופנים הבאים:
  + מקביליות – בתכנות פונקציונאלית, קריאה משורשרת לפונק' מונעות שינוי גלובלי למשתנים, ומאפשרת הרצה מקבילית בטוחה (דבר שדורש התאמה של הקוד בתכנות פרודצוראלית)
  + אופטימיזציות – תכנות פונקציונאלית דוגלת באבסטרקציה של הקוד (לדוג' שימוש בmap לעבור על איברי המערך, ולא בהכרח דרך לולאה), גישה שקשה למימוש בתכנות פרוצדוראלית.
  + אימות קוד – בעזרת פונק' מסדר גבוה, בגישה הפונק' מאפשרת בצורה קלה לאמת קוד וטיפוסי נתונים של קלטים. בתכנות פרוצדוראלית שמבוססת על הגדרת משתנים לאורך הפרוצדורות עלינו לבצע יותר בדיקות על מנת לאמת את תקינות הקוד

1. המרה לפרדיגמה הפונקציונאלית:

const pred: (x:number)=>Boolean=(x) =>x>60;

const averageGradesOver60: (grades:Array<number>) => number =

(grades) => grades.filter(pred).reduce((acc:number,cur:number)=>acc+cur,0)/(grades.filter(pred).length)

1. הגדרת טיפוסי נתונים ספציפים לפקודות הבאות:

(a) (x,y) => x.some(y)  // (x:Array<T>, y: (k:T) => boolean) result is type boolean

(b) x => x.reduce((acc,cur) => acc+cur,0) // acc:number, cur:number, x: Array<number> result is type number

(c) (x,y) => x ? y[0] : y[1] // x: boolean, y: Array<T> and y.length>=2, result is type T

(d) (f,g) => x => f(g(x+1)) // x: number, g = (number) => T1, f = T1 => T2, result is type function number => T2

1. הסבר לAbstraction Barriers

שימוש בקוד יכול להיות משתי נקודות מבט, של המתכנת ושל המשתמש. בעת יצירת קוד נדרש מאתנו (המתכנתים) לייצר קוד נכון לוגית ופונקציונאלי מספיק למשתמש. הרעיון הכללי של שכבות אבסטרקציה בתכנות מונחה עצמים הוא לספק מידע אינפורמטיבי על העצם למשתמש, על ידי שימוש בפונקציות ומשתנים עם שמות בעלי משמעות. סט הפעולות הזמינות למשתמש בעת שימוש בשכבות אטרקציה הוא ספציפי, כלומר, למשתמש אין גישה למימוש "מאחורי הקלעים" של המתכנת, והוא יכול להשתמש רק בסט הנתון לו. בשיטה זו אנו יוצרים מחסום למשתמש, ובכך גם מגנים על האובייקט מפעולות לא חוקיות וגם חוסכים ממנו את הצורך להבין ולדעת את מימוש מבנה האובייקט.