Part 1 : Theoretical Questions

שאלה 1:

1. שפות אימפרטיביות: שפה השמה דגש על רצפים של פקודות האומרות למחשב לשנות בהתאם את סביבת העבודה וביצוע הפקודות בזו אחר זו. לדוגמא Python,Java,C++.

2. שפות פרוצדורליות: שפה המאפשרת להגדיר קוד כפרוצדורה, כך שניתן לקרוא ממקומות שונים בקוד. בתכנות פרוצדורלי מחולקת תוכנית מחשב אחת לתת תוכניות רבות, שכל אחת מהן קרויה פרוצדורה וכל פרוצדורה יכולה לקרוא לפרוצדורה אחרת על-מנת לבצע פעולה שבה היא מתמחה. כל פרוצדורה בתוכנית פרוצדורלית מבצעת משימה מוגדרת, כחלק ממכלול המשימות המרכיב את התוכנית השלמה. לדוגמא Python,Java,C++.

3. שפות פונקציונליות: זו שפה ששמה דגש על חישוב ביטוי תוך שימוש בפונקציות ככלי ההפשטה העיקריים . הרצת תוכנית פונקציונלית היא חישוב של הביטוי/ם, כלומר מציאת הערך שלו ולא ביצוע רצף של פקודות. לדוגמא: Python,JS,Scheme,L1-L7.

השפה הפרוצדורלית לעומת השפה אימפרטיבית מגדילה את רמת האבסטרקטיות שבקוד, כלומר ברגע שמתאפשר לכתוב קוד המשתמש בקריאות לקטעי קוד במקומות שונים מאפשר גמישות, חיסכון בקוד וקריאות.

השפה הפונקציונלית לעומת השפה הפרוצדורלית לא משנה את נתוני הקלט, מאחר וכל המשתנים הם Immutable ויהיה קל יותר למקבל תהליכים או להוסיף אופטימיזציות. בתכנות פרוצדורלי יש חשיבות לסדר הפעולות(הפרוצדורות) לעומת תכנות פונקציונלי בה אין חשיבות מאחרת וכל פעולה אינה תלויה באחרת ולכן קל יותר.

שאלה 2:

(a) <T>(x: Array<T>, y: predicate(value : T, index : number, array: T[])) : boolean => x.some(y)

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

(c) <T>(x : boolean , y : Array<T>) : T => x ? y[0] : y[1]

שאלה 3:

מחסום הפשטה = Abstraction barrier

לכל פיסת תוכנה יש, או צריכה להיות, מחסום הפשטה המחלק את העולם לשני חלקים: לקוחות ומיישמים. הלקוחות הם אלה המשתמשים בתוכנה. המיישמים הם אלה שבונים אותו. הלקוחות אינם צריכים לדעת כיצד התוכנה עובדת. כדי לבצע את עבודתם, עליהם לסמוך על התוכנה. המיישמים אכן צריכים לדעת כיצד התוכנה עובדת. כמממש, אתה תמיד צריך להניח שיש באגים בתוכנה שאתה מיישם, ועליך לחפש אותם בשקידה באמצעות בדיקות. לכל פיסת תוכנה חשוב לדעת באיזה צד של מחסום ההפשטה אתה נמצא. האם אתה לקוח או מיישם? כאשר אתה מיישם תוכנה, אתה משתמש גם בתוכנות אחרות, כך שאתה מיישם תוכנה כלשהי אך לקוח של תוכנות אחרות. יתרה מכך, לעתים קרובות אתה משתמש בתוכנה שיישמת בעצמך. אתה לקוח כשאתה משתמש בו, אבל מיישם כשאתה מיישם אותו. מכיוון שלעתים קרובות מתכנתים עוברים קדימה ואחורה בין חלקים שונים בתוכנית שהם כותבים, עליהם להיות מסוגלים לעבור מלקוח למיישם ולחזור ללקוח.