**חלק 1: שאלות תאורטיות**

**שאלה 1:**

**שאלה 2:**

תכנית שניתן להריץ באופן מקבילי –

תכנית שלא ניתן להריץ באופן מקבילי –

**שאלה 3:**

כל תכנית ב ניתן לכתוב בשפה שכן האופרטור המיוחד define מאפשר לנו לפשט את הקוד, ניתן לקחת תוכנית ב- ובכל מקום שכתוב המשתנה שהגדרנו באמצעות הdefine, ניתן להחליף בהשמה שביצענו לאותו משתנה עד כה.

**שאלה 4:**

כל תכנית ב ניתן לכתוב בשפה שכן האופרטור המיוחד define מאפשר לנו לפשט את הקוד, ניתן לקחת תוכנית ב- ובכל מקום שכתוב המשתנה/פונקציה שהגדרנו באמצעות הdefine, ניתן להחליף בהשמה שביצענו לאותו משתנה/פונקציה עד כה.

**שאלה 5:**

את הפונקציות map, filter, all ניתן להריץ באופן מקבילי, שכן על כל איבר ברשימה אנחנו מבצעים פעולה נפרדת שלא תלויה בפעולות שמבצעים על יתר איברי הרשימה.

את הפונקציה reduce לא ניתן להריץ באופן מקבילי, שכן הפעולה מבצעת סכימה של איברי הרשימה, המבוססת על סכימת האיברים עד כה יחד עם האיבר הבא.

את הפונקציה compose לא ניתן להריץ באופן מקבילי, שכן הפעולה מרכיבה פונקציות לפי סדר מסוים, ולכן לא יכולה לרוץ במקביל.

**חלק 2: functions contracts**

**Q1**

; Signature : append(lst1 lst2)

; Type: [ List(T) \* List(U) -> List(T|U)]

; Purpose: concatenate list1 to list2

; Pre-conditions: lst1 and lst2 are lists

; Tests: (append '(1 2) '(3 4)) → '(1 2 3 4)

**Q2**

; Signature : reverse(lst)

; Type: [List(T)-> List(T)]

; Purpose: reverse the list members

; Pre-conditions: lst is a List

; Tests: (reverse '(1 2 3)) → '(3 2 1)

**Q3**

; Signature : duplicate-item(lst dup-count)

; Type: [List(T)\*List(Number)->List(T)]

; Purpose: duplicate each item of lst according to the number define in the same position in dup-count

; Pre-conditions: lst and dup-count are lists. dup-count elements are numbers and is not empty.

; Tests: (duplicate-items '(1 2 3) '(1 0))→ '(1 3)

(duplicate-items '(1 2 3) '(2 1 0 10 2))→ '(1 1 2)

**Q4**

; Signature : payment(n coins-lst)

; Type: [Number \* List(Number)->Number]

; Purpose: returns the number of possible ways to pay the money with these coins

; Pre-conditions: n is a positive Number, coins-lst is a list. all elements in coin-lst are positive

; Tests: (payment 10 ‘(5 5 10)) → 2

(payment 5 ‘(1 1 1 2 2 5 10) → 3

**Q5**

; Signature : compose-n( f n)

; Type: [func \* Number -> closure]

; Purpose: compose f n times

; Pre-conditions: f is a function, n is a Number.

; Tests: (define mul8 (compose-n (lambda (x) (\* 2 x)) 3))

(mul8 3) → 24