עבודה 2 – עקרונות שפות תכנות

שאלה 1.1:

צורות מיוחדות נדרשות בשפה על מנת להגדיר את מבנה השפה (תחביר וסמנטיקה) עבור ביטויים מורכבים שהם לא אופרטורים רגילים (ברירת מחדל) כפי שמוגדר באינטרפרטר.

כמו כן במקרה של אופרטור פרימיטיבי האינטרפרטר מחשב את הערכים של הביטויים לפי כללים (חוקים בשפה של האופרטור) , **אך לעומת זאת** במקרה של צורה מיוחדת הם נגזרות לפי כללים מיוחדים המוגדרים כמבנה בשפה. לדוגמא : ((define a (+ 5 2) הוא צורה מיוחדת שמחשבת את תת הביטוי של הערך (5+2) ומוסיפה אותו לסביבה (binding) כזוג סדור <a,7> , כלומר זה הוא חוק הגזירה של הצורה המיוחדת define כביטוי מורכב שהוא לא אופרטור רגיל (ברירת מחדל) בשפה.

שאלה 1.2:

תוכנית בL1 שניתן לבצע במקביל ע"י תרדים שונים :

$$(+7\ 2)$$
$$(-5\ 3)$$

תוכנית בL1 שלא ניתן לבצע במקביל ע"י תרדים שונים :

$$(define\ x\ 2)$$
$$(+\ x\ 3)$$

התרחיש הוא שתרד אחד התחיל את השורה הראשונה ואז נוצר context switch ותרד אחר מנסה לבצע את השורה השנייה אך הסביבה לא מכירה את המשתנה x ולכן תוכנית זו תיכשל.

שאלה 1.3:

בשפה L1 קיימת צורה מיוחדת אחת "define" ומהגדרת define L0 אינה נכללת בה ולכן אין בL0 אפ צורה מיוחדת. נשים לב שעבור כל תוכנית בL1 : אם אינה מכילה את הצורה המיוחדת define אז היא גם בL0 , ובמידה וכן מכילה , נוכל להמיר כל משתנה שבוצע לו define בערך שלו בכל מקום שיש בו שימוש. לדוגמא :

L1 :

- (define x 3)
- (+ x 2)

L0:

- (+ 3 2)

שאלה 1.4

בשפה L2 ישנם הצורות המיוחדות הבאות: if, lambda, define. השפה L20 היא השפה L2 ללא define כלומר הצורות המיוחדות שיש בה הן: if, lambda. נשים לב כי עבור התוכנית (כפי שראינו בהרצאה) :

```
(define fact-iter
  (lambda (n acc)
    (if (= n 1)
        acc
        (fact-iter (- n 1) (* n acc)))))
```

ללא שימוש בdefine לא ניתן לבצע רקורסית זנב מאחר והסביבה לא מכירה את fact-iter.

שאלה 1.5

Map: **ניתן להריץ מקבילית** - מאחר ופונקציה זו מפעילה פונקציה על כל איבר שונה במערך, אין תלות בין האיברים ולכן אין חשש לפגיעה גם כאשר מתקיימת ריצה מקבילית.

Reduce: **לא ניתן להריץ במקביל** - שכן בפונקציה זו ישנו ערך התחלתי acc אשר כל האיברים במערך תלויים בו וישנם פונקציות כגון חילוק אשר לא קומוטטיביות ועלולות להחזיר תוצאה שגויה.

Filter: **ניתן להריץ מקבילית** - מאחר ופונקציה זו מפעילה פרידיקט "סינון" על כל איבר שונה במערך, הרי שאם איבר עבר את פרידיקט "סינון" אזי הוא יעבור אותה בכל פעם באותו האופן. אין תלות בין האיברים ולכן אין חשש לפגיעה גם כאשר מתקיימת ריצה מקבילית.

All: **ניתן להריץ מקבילית -** מאחר ופונקציה זו מפעילה פרידיקט המחזיר #t\#f עבור כל איבר שונה במערך, שכן כל איבר יחזיר את אותו ערך החזרה עבור הפרידיקט תמיד. בסופו של דברים מדובר בפעולה לוגית של AND ולכן התוצאה הסופית לא תשתנה.

Compose: **לא ניתן להריץ במקביל –** הרכבת פונקציות איננה קומוטטיבית $f\big(g(x)\big) \neq g(f(x))$ ולכן קיימות פונקציות שהרצה מקבילית תיתן תוצאות שונות.

שאלה 1.6:

```
(define b 1)
(define c 2)
(define pair
  (lambda (a b)
    (lambda (msg)
      (if (eq? msg first)
        ((lambda () a) )
        (if (eq? msg second)
          ((lambda () b))
          (if (eq? msg f)
            ((lambda () (+ a b c)) )
            #f
          )
        )
      )
    )
  )
)

(define p34 (pair 3 4))
((lambda (c) (p34 `f)) 5)
```

התשובה המתקבלת היא **9**.

מכיוון שכאשר הבנאי של pair נקרא בdefine p34 , c מוגדר בclosure בתור 2 (בשורה השנייה מלמעלה). ולכן כאשר נקרא לבנאי הסביבה מכירה את c בתור 2.

לכן : a=3 , b=4 , c=2 כלומר סכומם בקריאה לפונקציה f הינו 9.

Contracts

Q2.1

; Signature: append(lst1,lst2)
; Type: [(List(T1)*List(T2) -> List(T1|T2)]
; Purpose: gets 2 lists and returns their concatenation.
; Pre-conditions: none
; Tests: (append '(1 2) '(3 4)) -> '(1 2 3 4) ; (append '("hello") '(3 4)) -> '("hello 3 4)

Q2.2

; Signature: reverse(lst)
; Type: [(List(T) -> List(T)]
; Purpose: gets a list and reverses it.
; Pre-conditions: none
; Tests: (reverse '(1 2 4)) -> '(4 2 1)

Q2.3

; Signature: duplicate-items(lst,dup-count)
; Type: [(List(T)*List(Number) -> List(T)]
; Purpose: gets 2 lists and returns a list with duplicates of each item in lst according to the number defined in the same position in dup-count list.
; Pre-conditions: dup-count contains numbers and is not empty.
; Tests: (duplicate-items '(1 2) '(3 2)) -> '(1 1 1 2 2) ;
(duplicate-items '(1 2 3) '(2 0)) -> '(1 1 3 3);
(duplicate-items '(1 4) '(2 1 4 5)) -> '(1 1 4);

Q2.4

; Signature: payment(n,coins-lst)
; Type: [(Number*List(Number) -> Number]
; Purpose: gets a sum of money and list of available coins and returns the number of possible ways to pay the money with this coins.
; Pre-conditions: $n \geq 0$
; Tests: (payment 10 '(5 5 10)) -> 2 ; (payment 5 '(1 1 1 2 2 5 10)) -> 3;

Q2.5

; Signature: compose-n(f,n)
; Type: [(T -> T)*Number-> (T -> T)]
; Purpose: gets an unary function f and a number n and return the closure of the n-th self-composition of f.
; Pre-conditions: $n > 0$
; Test: ((compose-n (lambda(x) (+ 2 x)) 2) 3) -> 7
((compose-n (lambda(x) (* 2 x)) 2) 3) -> 12