# עבודה 2 בעקרונות שפות תכנות – שאלה 1

## Q1.1

|  |  |
| --- | --- |
| ביטוי אטומי פרמיטיבי: | #t |
| ביטוי אטומי לא-פרמיטיבי | a (בתור VarRef) |
| ביטוי מורכב לא-פרמיטיבי | (+ 1 (+ 1 1)) |
| ערך אטומי פרמיטיבי | 1 |
| ערך אטומי לא-פרמיטיבי | <Closure (x) (\* x x)> |
| ערך מורכב לא-פרמיטיבי | '(2 . 3) |

## Q1.2

"Special form" הוא ביטוי מורכב שלא מחושב (evaluated) כמו ביטוי מורכב רגיל. הוא דורש חוקים מיוחדים לחישובו.

דוגמה לביטוי special form:

(define x 2)

לו היינו מחשבים את הביטוי הזה בדרך הסטנדרטית אז המשתנה x היה צריך להיות מחושב – אך זו היתה שגיאה (כי הוא עדיין לא מוגדר, אנו מגדירים אותו באמצעות החישוב המיוחד הזה שטרם בוצע). ולכן חוק החישוב של define הוא ראשית לחשב את 2 ואז לבצע binding בינו לבין השם x.

## Q1.3

משתנה חופשי הוא משתנה שלא מקושר לערך בתוך ה-scope בו הוא נמצא. למשל בביטוי:

(lambda (x) (\* x y))

המשתנה y הוא משתנה חופשי (בעוד שהמשתנה x הוא משתנה bound).

## Q1.4

S-Expression הוא דרך לייצג רשימות מקוננות כעץ, בעזרת סוגריים מאוזנים. הוא מוגדר רקורסיבית בעזרת ביטוי אטומי כמקרה בסיס, או ביטוי מהצורה (x y) כאשר x,y הם S-Expressions (ייתכן גם מספר תתי-ביטויים הגדול מ-2).

דוגמה ל- S-Expression:

(+ 2 2)

## Q1.5

Syntactic abbreviation הוא מבנה תחבירי המקביל סמנטית למבנה תחבירי אחר. מעין שתי דרכים תחביריות ל

דוגמה לכך היא let expression, שמתורגם פנימית בשפה למבנה של lambda, שמשמעותו הסמנטית מקבילה ושעליו מופעל חוק חישוב שכבר קיים.

נמחיש: הביטוי

(let ( (<var1> <exp1>) (<var2> <exp2>) ... (<varn> <expn>) )<body>)

מקביל לביטוי

( (lambda (<var1> ... <varn>) <body>)<exp1> ... <expn> )

דוגמה נוספת – cond שהוא Syntactic abbreviation ל- if. נמחיש. הביטוי:

(cond (<test> <then>)  
      (else <else>))

מקביל לביטויי

(if <test> <then> <else>)

## Q1.6

כל תוכנית ב- L3 ניתנת להמרה לתוכנית מקבילה ב-L30. נסביר כיצד.

בהינתן רשימה ב-L3 ניתן לייצג אותה באמצעות זוגות (pairs) מקוננים ב-L30.

יצירת רשימה ב-L3 באמצעות (list <var1> <var2> … <varn>) ויצירת זוגות מקוננים ב-L30 באמצעות (cons <var1> (cons <var2> (… cons <varn> '() ) … )) מניבה את אותו הערך '(<var1> <var2> … <varn>)

כמו כן אין שינוי בהתנהגות הפונקציות car ו-cdr.

## Q1.7

להשלים.

## Q1.8

אם נממש map כך ששישמור על הסדר המקורי ברשימה המוחזרת אך יעבור על האיברים בסדר הפוך נקבל את אותה הרשימה (ביחס למימוש רגיל של map). זאת משום שהפונקציה שניתנת ל-map מופעלת על כל איבר ברשימה שניתנה ל-map באופן עצמאי, שלא תלוי בשאר איברי הרשימה או במיקום האיבר ברשימה.

באופן דומה, אם נממש filter כך שישמור על הסדר המקורי ברשימה המוחזרת אך יעבור על האיברים בסדר הפוך נקבל את אותה הרשימה (ביחס למימוש רגיל של filter). זאת משום שהפרדיקט שניתן ל-filter מופעל על כל איבר ברשימה שניתנה ל-filter באופן עצמאי, שלא תלוי בשאר איברי הרשימה או במיקום האיבר ברשימה.

עם זאת, אם נממש reduce כך שיעבור על האיברים בסדר הפוך – לא בהכרח נקבל את אותה התוצאה (ביחס למימוש רגיל של reduce). דוגמה לכך:

אם נגדיר

(define fun

 (lambda (x y)

    (if (> y 5) (\* x y) (+ x y))))

אז (reduce fun 3 '(4 5)) תחזיר 32 אבל (reverse-reduce fun 3 '(4 5)) תחזיר 35.

לגבי compose – גם כאן יש משמעות לסדר המעבר על האיברים ברשימת הקלט, כלומר אם נממש compose כך שיעבור על האיברים בסדר הפוך – לא בהכרח נקבל את אותה התוצאה (ביחס למימוש רגיל של compose). דוגמה לכך:

אם נגדיר

(define add2 (lambda (x) (+ 2 x)))

(define mul2 (lambda (x) (\* 2 x)))

אז ((compose '(add2 mul2)) 3) תחזיר 8 אבל  
((reverse-compose '(add2 mul2)) 3) תחזיר 10.

# שאלה 2

Signature: map(func,lst)

Purpose: Apply func to all elements in lst and return the list of the results

Type: [ (T1->T2) \* List(T1) -> List(T2)]

Example: map((lambda (x) (+ 2 x)), '(1 2 3) ) should return '(3 4 5)

Pre-conditions: None

Post-condition: '(func(lst[0]) func(lst[1]) ... func(lst[n-1]))

Tests: (map (lambda (x) (+ 2 x)) '(1 2 3) )) ==> '(3 4 5)

Signature: reduce(reducer, init, l)

Purpose: Combine all the values of l using reducer

Type: [(T1 \* T2 -> T2) \* T2 \* List(T1) -> T2]

Example: (reduce + 0 '(1 2 3)) --> (+ 1 (+ 2 (+ 3 0)))

Pre-conditions: None

Post-condition: None

Tests: (reduce + 0 '(1 2 3)) ==> 6

Signature:length-of-lst(lst)

Purpose: calculate the length of the given list

Type: [List(T) -> Number]

Pre-conditions: None

Post-condition: None

Tests: (length-of-lst (list 1 2 3)) → 3

Signature: last-element(lst)

Purpose: Return the last element of a given list

Type: [List(T) -> T]

Example: (last-element (list 1 3 4)) --> (car (cdr (cdr '(1 3 4))))

Pre-conditions: lst is not empty

Post-condition: None

Tests: (last-element (list 1 3 4)) ==> 4

Signature: power(n1, n2)

Purpose: Return n1 to the power of n2 (n1^n2)

Type: [number \* number -> number]

Example: (power 2 4) -> (\* 2 (\* 2 (\* 2 2)))

Pre-conditions: n1 and n2 are non-negative numbers

Post-condition: None

Tests: (power 2 4) ==> 16

Signature: sum-lst-power(lst, n)

Purpose: apply power on each element of lst, and then sum the results

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

Example: (sum-lst-power (list 1 4 2) 3) → 1^3+ 4^3 + 2^3 = 73

Pre-conditions: n >= 0

Post-condition: Result = (+ (power lst[0] n) (+ ... powerlst[n]) ... )

Tests: (sum-lst-power (list 1 4 2) 3) ==> 1^3+ 4^3 + 2^3 = 73

Signature: remove-last-element(lst)

Purpose: Return a copy of the given list without its last element

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

Example: (last-element (list 1 3 4)) --> (cons 1 (cons 3 (cons 4 '())))

Pre-conditions: None

Post-condition: None

Tests: (remove-last-element (list 1 3 4)) ==> '(1 3)

Signature: num-from-digits(lst)

Purpose: Return the number consisted from lst's digits

Type: [List(number) -> number]

Pre-conditions: All of the numbers in lst are not negative

Post-condition: None

Tests: (num-from-digits (list 2 4 6)) ==> 246

Signature: is-narcissistic(lst)

Purpose: Find out if a number is narcissistic (equals to the sum of

its digits each raised to the power of the number of its

digits)

Type: [List(number) -> boolean]

Pre-conditions: All of the numbers in lst are not negative

Post-condition: None

Tests: (is-narcissistic (list 1 5 3)) → #t

Tests: (is-narcissistic (list 1 2 3)) → #f