תאריך הבוחן: 19.5.2017 שעה: 9:00
שם המורה: פרופ' מיכאל אלחדד, פרופ' מירה בלבן, ד"ר דנה פיסמן, ד"ר מני אדלר, ד"ר ירון גונן.
בוחן ב: עקרונות שפות תכנות
מס. הקורס: 202-1-2051
מיועד לתלמידי: מדעי המחשב, הנדסת תוכנה, מתמטיקה ומדעי המחשב
שנה: ב' סמסטר: ב'
משך הבוחן: 2 שעות

שאלה 1 Applicative/normal operational semantics

נתונה התוכנית הבאה בשפת **Scheme**.

```
(define y 7)
(define double (lambda (x) (display x) (newline) (* x 2)))
(define g (lambda (x) (if (= y (double y)) 'eq 'neq)))
(g (double 1))
```

א. [**6** נק']

מה יודפס ומה יהיה הערך שיוחזר בשערוך התוכנית ע"י applicative-eval?

The returned value is 'neq (of type Symbol)
The following will be printed to the screen:
1
7

ב. [**6** נק']

מה יודפס ומה יהיה הערך שיוחזר בשערוך התוכנית ע"י normal-eval?

The returned value is 'neq (of type Symbol)
The following will be printed to the screen:
7

שאלה 2 – Lexical Address

נתון הביטוי הבא בשפת **Scheme**.

```
((lambda (x y)
    (display x)
    (newline)
    (display y)
    (newline)
    ((lambda (x z)
        (newline)
        (let ((y 3) (z x))
              (if (> y x)
                  (display y)
                  (display z))))
     x y)) 1 9)
```

א. [**4** נק'] לכל מופע של אחד מתתי הביטויים בקבוצה **{x,y,z}** רשום/רשמי את סוגו (var-ref, var-decl).

vd = var-decl
vr = var-ref

```
((lambda (x vd  y vd )
  (display x vr )
  (newline)
  (display y vr )
  (newline)
  ((lambda (x vd  z vd )
      (newline)
      (let ((y vd  3) (z vd  x vr ))
            (if (> y vr  x vr )
                (display y vr )
                (display z vr ))))
   x vr  y vr )) 1 9)
```

ב. [**6** נק'] לכל מופע של אחד מתתי הביטויים בקבוצה **{x,y,z}** מסוג var-ref ציין/י את ה lexical-address שלו. (למופעים מסוגים אחרים רשום "non var-ref").

להזכירך המבנה של **lexical address** הינו **[x : m n]** כאשר **m** הוא המרחק ל **scope** בו מוגדר המשתנה **x** ו **n** הוא המיקום של **x** ברשימת הפרמטרים.

NVR = non var-ref

```
((lambda (x NVR  y NVR )
  (display x [0 0] )
  (newline)
  (display y [0 1] )
  (newline)
  ((lambda (x NVR  z NVR )
      (newline)
      (let ((y NVR  3) (z NVR  x [0 0] ))
            (if (> y [0 0]  x [1 0] )
                (display y [0 0] )
                (display z [0 1] ))))
   x [0 0]  y [0 1] )) 1 9)
```

ג. [**6** נק'] בביטוי הבא בשפת Scheme מצויינים **ה** lexical addresses של המשתנים המופיעים **כ**var-decl , אבל שמות המשתנים מחוקים.

הוסף/הוסיפי לביטוי שמות משתנים שיתאימו ל lexical-addresses המצויינים.

```
> (define foo
     (lambda (_x _y)
         (- (+ [_x : 0 0]
               ((lambda (_z) (* [_z : 0 0] [_x : 1 0]))
                [_y : 0 1]))
         [_y : 0 1])))
> (foo 5 2)
13
> (foo 10 7)
73
```

Note: z could be consistently renamed by y,
but not by x

שאלה 3 [6 נק'] – Abstract Syntax Tree

כתוב/כתבי את הביטוי שיתקבל ע"י הפעלת ה Parser על הביטוי הבא :

```
((lambda (x) (if (> x 0) x 'neg)) 5)
```

```
'(app-exp (proc-exp ( (var-decl x) )
                    ( (if-exp ( app-exp (var-ref >)
                                        ( (var-ref x) (num-exp 0) ) )
                              (var-ref x)
                              (lit-exp neg) ) ) )
          ( (num-exp 5) ) )
```

שאלה 4 – Substitution and Renaming

נתון הביטוי הבא בשפת **Scheme**.

```
(define bar (lambda (x)
  (display x)
  (newline)
  ((lambda (x)
      (display (list x y))
      (newline)
      (let ((x 1) (y 2))
            (if (> y x)
                (display y)
                (display x))))
   x)))
(bar 5)
```

א. [**4** נק'] בצע **Renaming** לביטוי כך שלכל המשתנים המופיעים כ**var-decl** יהיה שם שונה. רשום/רשמי את הביטוי המתקבל:

```
(define bar (lambda (x_1)
  (display x_1)
  (newline)
  ((lambda (x_2)
      (display (list x_2  y ))
      (newline)
      (let ((x_3 1) (y_1 2))
            (if (> y_1 x_3)
                (display y_1)
                (display x_3))))
   x_1)))
(bar 5)
```

ב. [**4** נק'] בצע את ה **Substitution** המתבקש לאור שערוך הביטוי **(bar 5)**.
רשום/רשמי את הביטוי המתקבל:

```
(display 5)
(newline)
((lambda (x_2)
  (display (list x_2 y))
  (newline)
  (let ((x_3 1) (y_1 2))
   (if (> y_1 x_3)
      (display y_1)
      (display x_3))))
 5)))
```

Note 1: The substitution that happens due to wanting to evaluate (bar 5) does the above mentioned substitution, it does not go deeper.
The deeper substitutions happen for the sake of evaluating the inner application of "(lambda (x_2) (display (list ...)))" on 5, and not the application (bar 5).

Note 2: The let is inside a lambda, so the variables it defines as well are not substituted.

Note 3: The substitution result does not substitute the variable of the syntactic lambda declaration "lambda (something)," rather it eliminates "lambda (something)" and yields only a body with the parameters substituted, as seen on the left in the solution.

שאלה 5 – Functional Abstractions

א. [**6** נק'] הפונקציה **high-some(pred)** מחזירה פונקציה שמקבלת רשימה ומחזירה את הערך הבוליאני **true** אם לפחות אחד האיברים ברשימה מקיים את הפרדיקט **pred**.
לדוגמה

```
> high-some(even)([1,2,3])
true
> high-some(prime)([10,20,30])
false
```

--- כתוב/כתבי מימוש לפונקציה **high-some** בשפת התכנות **Typescript**.

--- השתמש/י ב **functional abstractions** שלמדנו בכיתה **map, filter, reduce**.

--- ציין/י את ה **full-type** של הפונקצייה.

function high-some (

```
Type: [[T => Boolean] => [Array<T> => Boolean]]

let highSome = function (pred){
    return function(array)
        {return array.reduce((acc,cur) => acc || pred(cur)), false) }
  }
```

ב. [**6** נק'] הפונקציה **high-every(pred)** מחזירה פונקציה שמקבלת רשימה ומחזירה את הערך הבוליאני **true** אם כל אחד מהאיברים ברשימה מקיים את הפרדיקט **pred**.
לדוגמה

```
> high-every(even)([1,2,3])
false
> high-every(prime)([2,3,5])
true
```

--- כתוב/כתבי מימוש לפונקציה **high-some** בשפת התכנות **Scheme**.

--- השתמש/י ב **functional abstractions** שלמדנו בכיתה **map, filter, foldr**.

--- ציין/י את ה **full-type** של הפונקצייה.

(define high-every

```
Type: [[T => Boolean] => [list(T) => Boolean]]

(define high-every
   (lambda(pred)
     (lambda(lst)
       (foldr (lambda (x acc) (and acc (pred x))) #t lst))))
```

Common mistake in Q6A and Q6B:
implementation of a function that gets (pred,list) and return Boolean
instead of getting (pred) and return a function that gets (list) and return Boolean

שאלה 6 – Reduce

א. [**6** נק'] ממש/י ב **Typescript** פונקצייה שמחזירה את מכפלת כל המספרים במערך (רשימה) נתון בעזרת **reduce**.
--- ציין/י את ה **full-type** של הפונקצייה.
לדוגמה

```
> product([1,2,3])
6
```

```
import * as R from 'ramda';

function product(arr: number[] ): number{
    return R.reduce((acc, cur) => acc*cur, 1, arr);
}
```

ב. [**4** נק'] מה צריך להיות הערך של **product ([])**?
נמק/י

Should be the initial value passed, 1.

It is the identity element of the preformed operation, here, product.

Common mistakes: - 0 instead of 1

ג. [**4** נק'] ממש/י ב **Typescript** פונקצייה שממשת את פעולת "עצרת" ע"י שימוש ב **range** מספריית **ramda** וב **product** מהסעיף הקודם.
--- ציין/י את ה **full-type** של הפונקצייה.

```
> factorial(3)
6

> import * as R from 'ramda';

R.range(1,5)

[ 1,2,3,4 ]
```

```
function factorial(n: number): number{
  return product(R.range(1,n+1));
}
```

ד. [**6** נק'] ממש/י ב **Typescript** את הפונקצייה **map(.)** על ידי הפונקצייה **reduce(.)**

--- ציין/י את ה **full-type** של הפונקצייה.

```
import * as R from 'ramda';

function map <T1,T2>(f: [x:T1 => T2], arr: T1[]): T2[] {
   return R.reduce((acc,curr) => acc.concat(f(curr)), [], arr)
}
```

Common mistakes:
- incorrect type
- using 'push' (returns void) instead of 'concat'

ה. [**6** נק'] ממש/י ב **Typescript** את הפונקצייה **filter(.)** על ידי הפונקצייה **reduce(.)**

--- ציין/י את ה **full-type** של הפונקצייה.

```
import * as R from 'ramda';

function filter <T1>(f: [x:T1 => boolean], arr: T1[]): T1[]{
  return R.reduce((acc,curr) => f(curr) ? acc.concat(curr) : acc, [], arr)
}
```

Common mistakes:
- incorrect type
- using 'push' (returns void) instead of 'concat'

שאלה 7 – Typing

Note: any two of the arguments below were accepted

א. [**4** נק'] תאר/י שני יתרונות פוטנציאלים לשפות תכנות שהן **Untyped**

* כתיבת הקוד וקריאתו לעיתים פשוטים יותר ללא type annotations שמסרבלים את הקוד.

* patching
קלות עדכון קוד קיים לאור שינוי הspec. למשל אם רוצים להוסיף לפעמים שדה מסויים ל type קיים שנעשה בו שימוש רב לאורך קוד גדול בשפות untyped ניתן לבצע את השינוי בקלות ע"י שינוי ה type ושינוי רק המתדות שצריכות להתייחס לשדה החדש. בעוד שבשפות typed יהיה צורך לפעפע את השינוי ולשנות בהרבה יותר מקומות.

* כתיבת קוד גנרי יותר מבלי להצטרך להגדיר templates כמו בשפות typed כשרוצים לכתוב בהם פונקציות גנריות שעובדות על טיפוסים שונים באשר הם מבלי להתייחס לסוג הטיפוס למשל פעולות על רשימות או עצים שבהם תוכן התא לא חשוב (חישוב אורך רשימה, עומק עץ וכו').

ב. [**4** נק'] מה ההבדל בין **nominal typing** ל- **structural typing**?

ב Structural Typing ההחלטה האם טיפוס A הוא תת טיפוס של טיפוס B נקבעת לפי המבנה של הטיפוסים בלבד: אם טיפוס A מגדיר קבוצת ערכים שמוכלת בקבוצת הערכים שטיפוס B מגדיר אז A יהיה תת טיפוס של B.

לעומת זאת בNominal Typing הקוד צריך לקשר באופן קונקרטי בין שני הטיפוסים כדי שאחד יוגדר להיות תת-טיפוס של השני. לדוגמה בשפת Java השימוש ב"extend" גורר שהמחלקה היורשת (המורחבת) היא תת-טיפוס של המחלקה ממנה יורשים (אותה מרחיבים).

ג. [4 נק'] האם יכולים להיות שני **types** כך שאחד הוא **subtype** של השני ב- **Typescript** אך לא ב - **Java** . אם כן תן/י דוגמה. אם לא נמק/י.

כן, למשל טיפוס A להלן הוא תת טיפוס של B להלן:

interface B { name: string};

interface A { name: string, age: number};

ד. [4 נק'] האם יכולים להיות שני **types** כך שאחד הוא **subtype** של השני ב- **Java** אך לא ב -**Typescript** . אם כן תן/י דוגמה. אם לא נמק/י.

לא, מכיוון שבJava כדי ש טיפוס A יהיה תת-טיפוס של טיפוס B הוא צריך להיות מוגדר ע"י extend שלו.
ואז, הוא אוטומטית יורש את כל השדות של B ולכן יהיה תת-טיפוס של B גם מבחינה מבנית.

Note: This operation is referred to as "disjoint union".
The obtained set should include all elements of both sets, thus its cardinality (size) is the sum of the cardinalities of the given sets.
The second component, "a tag" (here 0 or 1) is added to the elements to distinguish the set of origin.

ה. [4 נק'] השלם למה שווים הביטויים הבאים:

$\{a, b\} \uplus \{x, y\} =$ **{(a,0), (b,0), (x,1), (y,1) }**

$\{a, b, c, d\} \uplus \{c, d, e\} =$ **{(a,0), (b,0), (c,0), (d,0) (c,1), (d,1), (e,1) }**

ו.

ו. [4 נק'] הוסף/הוסיפי הצהרות של **Types** לתוכניות הבאות ב **Typescript** :

```
const d2 : number => number
      = (x) => (2*x)

const Church: (number => number) * number => (number=> number)
     = (f, n) => n === 0 ? (x) => x :
                           (x) => f(Church(f,n-1)(x))

Church(d2,8)(1)
```

```
import {map} from 'ramda';

interface Date {
    year : ___number___;
    month : ___number___;
    day : ___number___;
}

interface person {
    birthDate : ___Date___;
    name : ___string___;
}

const Year : ___Date => number___ =
      date => date.year;

const computeAges : ___Person[] => number[]___ =
      persons => map(p=>2017-p.birthDate.year, persons);

{
    let persons = [{name:"avi", birthDate:{year:1990, month:7, day:10}},
                   {name:"batia", birthDate:{year:1994, month:3, day:2}}];
    computeAges(persons)
}
```

Note: other answers such as string or a list of 12/7 values for month/day were also accepted.