תאריך 02.07.2018
שם המרצים: מני אדלר, מיכאל אלחדד, ירון גונן
מבחן בקורס: עקרונות שפות תכנות
קורס' מס: 202-1-2051
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה
שנה: ב' סמסטר: ב'
משך הבוחן: 3 שעות
חומר עזר: אסור

הנחיות כלליות:

1. ההוראות במבחן מנוסחות בלשון זכר, אך מכוונות לנבחנים ולנבחנות כאחד.
2. מבחן הכתוב בעיפרון חלש המקשה על הקריאה, לא יבדק
3. יש לענות על כל השאלות בגוף המבחן בלבד (בתוך השאלון). מומלץ לא לחרוג מהמקום המוקצה.
4. אם אינך יודע את התשובה, ניתן לכתוב "לא יודע" ולקבל 20% מהניקוד על הסעיף/השאלה.

שאלה 1: lazy lists, AST _____ 25 נק
שאלה 2: ייצוג אובייקטים ב L, closures___ 20 נק
שאלה 3: הסק טיפוסים_____ 15 נק
שאלה 4: CPS _____ 26 נק
שאלה 5: תכנות לוגי _____ 20 נק
סה"כ_____ 106 נק

שאלה 1 - lazy lists, AST (25 נק)

א. ניתנה הגדרת ה-data type של lazy-lists כפי שהוגדרה בכיתה.
השלמו את ההגדרה של הפרוצדורה take - התייחסו למקרים שהרשימה קצרה מ-k איברים. **(3 נק)**

```
;; Lazy-list functional interface
(define cons-lzl cons)
(define empty-lzl empty)
(define head car)
(define tail
  (lambda (lzl)
    ((cdr lzl))))

;; Example lazy list
(define integers-from
  (lambda (n)
    (cons-lzl n (lambda () (integers-from (+ n 1))))))
(define integers (integers-from 0))

; Purpose: Return the first k elements of a lazy-list as a list
; Signature: take(lzl, k)
; Type: ____________________________________________
(define take
  (lambda (lzl k)
    (if ____________________________________________

        ____________________________________________

        ____________________________________________)))
```

Test 1: רשימה מספיק ארוכה
(take _____) =>

_____

Test 2: רשימה קצרה
(take _____) =>

_____

ב. הגדרו פרוצדורה **take-pad** שמקבלת lazy-list lzl, מספר k וערך כלשהו pad ומחזירה רשימה של בדיוק k איברים המכילה את האיברים הראשונים של lzl ובמידה ש-lzl קצרה את הערך pad עד לאורך k **(2 נק)**

```
(define short-lzl (cons-lzl 0 (lambda () empty-lzl)))
(take-pad short-lzl 3 '*)
→ '(0 * *)
(take-pad integers 3 '*)
→ '(0 1 2)

; Purpose: take exactly k elements from lzl with padding
; Signature: take-pad(lzl, k, pad)

; Type: ____________________________________________________
(define take-pad
  (lambda (lzl k pad)

    ____________________________________________________

    ____________________________________________________

    ____________________________________________________

    ____________________________________________________
```

ג. הגדרו פרוצדורה **consecutive-pairs** שמקבלת lazy-list ומחזירה lazy-list של זוגות איברים רציפים מהקלט ללא חפיפה. במקרה שרשימת הקלט מסתיימת, השלמו את הזוג האחרון עם ערך מיוחד למילוי (padding). **(4 נק)**

```
(take(consecutive-pairs integers '*) 4)
→ '((0 1) (2 3) (4 5) (6 7))
(take (consecutive-pairs short-lzl '*) 4)
→ '((0 *))
```

```
; Purpose: compute lazy list of pairs of elements from lzl
; Signature: consecutive-pairs(lzl, padding)

; Type: ____________________________________________________
(define consecutive-pairs
    (lambda (lzl padding)

     ____________________________________________________

     ____________________________________________________

     ____________________________________________________

     ____________________________________________________

     ____________________________________________________
```

ד. הגדר את הפרוצדורה ngram שמקבלת lazy-list ומחזירה רצפים של n איברים עם חפיפה.
במידה וה-lazy-list קצרה, כל האיברים של הרשימה חייבים להופיע בתוצאה - עם padding כנדרש. **(4 נק)**

```
(take (ngram integers 4 '*) 3)
→ '((0 1 2 3) (1 2 3 4) (2 3 4 5))
(take (ngram short-lzl 3 '*) 3)
→ '((0 * *))

; Purpose: Compute lazy list of ngrams from lzl
; Signature: ngram(lzl, n, pad)

; Type: ____________________________________________________
(define ngram
    (lambda (lzl n pad)

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________

        ____________________________________________________
```

ה. AST (12 נק)

ברצוננו להגדיר special-form חדש להקל על בניית lazy-lists. נגדיר את הביטוי החדש בשפה **make-lzl**:

```
(define lzl1 (make-lzl 1 empty-lzl))
;; equivalent to (cons-lzl 1 (lambda () empty-lzl))

(define lzl2 (make-lzl 2 lzl1))
;; equivalent to (cons-lzl 2 (lambda () lzl1))

(define integers (lambda (n) (make-lzl n (integers (+ n 1)))))
```

ניתן חלק מדקדוק השפה:

```
<cexp> ::= <number>        / NumExp(val:number)
  | <boolean>              / BoolExp(val:boolean)
  | <string>               / StrExp(val:string)
  | <prim-op>              / PrimOp(op:string)
  | <var-ref>              / VarRef(v:string)
  | (lambda (<var-decl>*) <TExp>* <cexp>+ ) /
        ProcExp(params:VarDecl[], body:CExp[], returnTE: TExp))
  | (if <cexp> <cexp> <cexp>) / IfExp(test: CExp, then: CExp, alt: CExp)
  | (<cexp> <cexp>*)          / AppExp(operator:CExp, operands:CExp[]))
```

השלמו את הדקדוק כדי לתמוך בביטוי מסוג **make-lzl** (BNF ו-abstract syntax) (3 נק)

כמו כן הגדרו את ה-type הנדרש ב-AST: (3 נק)

```
interface MakeLzlExp ____________________________________________

const makeMakeLzlExp = ____________________________ =>

    ____________________________________________________________;

const isMakeLzlExp = (x: any): ____________________________________;
```

השלם את הפרוצדורה שמשכתבת ביטוי מסוג **make-lzl** לביטוי אקוויוולנטי עם **cons-lzl (6 נק)**

```
// Purpose: rewrite a single makeLzl exp as a cons-lzl application
//          with nested lambda
// Signature: rewriteMakeLzl(makeLzlExp)

// Type: ____________________________________________________

const rewriteMakeLzl = (e: MakeLzlExp): ______________________ =>

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________

    ____________________________________________________________;
```

שאלה 2: ייצוג אובייקטים ב L (20 נק)

א. מה הפלט של הקוד הבא: **(2 נקודות)**

```
(define make-equation
  (lambda (a b)
    (lambda (msg)
      (cond ((eq? msg 'left) a)
            ((eq? msg 'right) b)))))

(let ((p1 (make-equation ‘var1 ‘var2))
      (p2 (make-equation ‘var3 ‘var2)))
   (eq? (p1 'left) (p2 'right)))
```

_____

נתון הממשק הבא ב TypeScript לייצוג אובייקט של סטודנט:

```
interface Student {
    tag: "student";
    id:number;
    name:string;
    grades:number[]
};
const makeStudent = (i:number, n:string, gs:number[]): Student
=>
    ({tag: "student", id:i, name:n, grades:gs});
const isStudent = (s: any): s is Student => s.tag === "student";
const getId = (s: Student): number => s.id;
const getName = (s: Student): string => s.name;
const getGrades = (s: Student): number[] => s.grades;
```

במהלך הקורס ראינו שתי דרכים לממש 'אובייקט' בשפות ה L שפיתחנו: כרשימה, וכ-Closure.

ב. באיזו שפה מבין L1-L5 ניתן כבר לממש אובייקט כרשימה (=השפה האפשרית הפשוטה ביותר), ובאיזו ניתן לממש אובייקט כ-closure? נמק **(2 נקודות)**

כרשימה:_____

כ-closure: _____

ג. השלימו את החסר במימוש הבא של האובייקט Student בשפה 5L. יש לתייג, אם אפשר, את הטיפוסים השונים **(10 נקודות)**

מימוש סטודנט כרשימה:

```
(define make-student-l
   (lambda ((i _________)
            (n _________)
            (gs ________)) _______________
     ____________________________________________________))

(define student?-l
   (lambda ((s ________)) __________
     ____________________________________________________))

(define student->id-l
   (lambda ((s ________)) __________
     ____________________________________))

(define student->name-l
   (lambda ((s ________)) __________
     ____________________________________))

(define student->grades-l
   (lambda ((s ________)) __________
     ____________________________________))
```

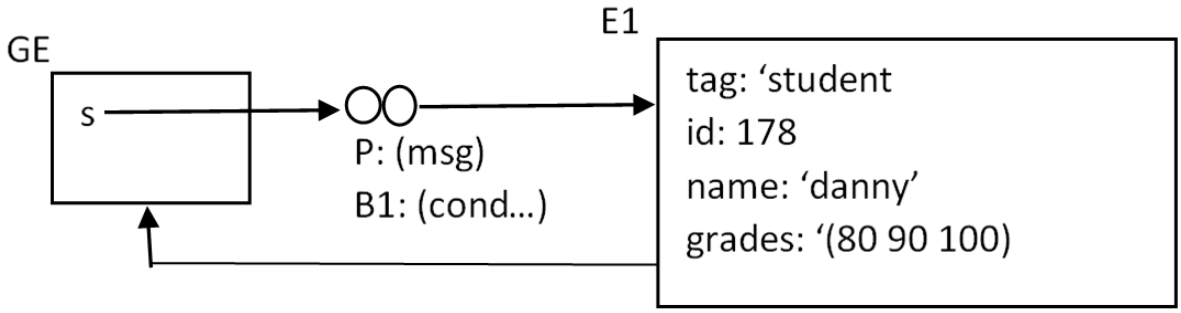
מימוש סטודנט כ-**closure**:

```
(define make-student-c
   (lambda ((i ________)
            (n ________)
            (gs _________)) ___________________________
     ____________________________________________________
     ____________________________________________________
     ____________________________________________________
     ____________________________________________________
     ____________________________________________________
))
```

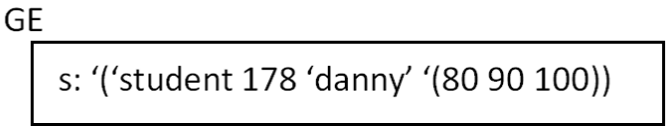
```
(define student?-c
   (lambda ((s ________)) __________
     ____________________________________________________))

(define student->id-c
   (lambda ((s ________)) __________
     ____________________________________))

(define student->name-c
   (lambda ((s ________)) __________
     ____________________________________))

(define student->grades-c
   (lambda (s ________) __________
     ____________________________________))
```

ד. השלימו את דיאגרמת הסביבות עבור ההרצה של שני קטעי הקוד הבאים (האחד בגישת רשימה, והשני בגישת closure) באינטרפרטר של 5L הממומש ע"פ מודל הסביבות: **(6 נקודות)**

```
(define s (make-student-c 178 "danny" '(80 90 100)))
(student->grades-c s)
```




```
(define s (make-student-l 178 "danny" '(80 90 100)))
(student->grades-l s)
```



שאלה 3: הסק טיפוסים (15 נקודות)

אלגוריתם הסק טיפוסים שנלמד בכיתה מייצר, בין היתר, משוואות טיפוסים לביטוי הנתון.
כזכור, המשוואות נגזרות מהסוג התחבירי של הביטויים: ביטויים פרימיטיביים (מספרים, ביטויים בוליאניים, אופרטורים פרימיטיביים), הגדרת פרוצדורה, הפעלת פרוצדורה/אופרטור, מבנה תנאי.
נתון הביטוי:

```
(
 (lambda (x y z) x y z)
 (f a)
 (* a b)
 c
)
```

ונתונים משתני הטיפוס לכל אחד מתתי הביטויים:

```
T1:   ((lambda (x y z) x y z) (f a) (* a b) c)
T2:   (lambda (x y z) x y z)
T3:   (f a)
T4:   (* a b)
T5:   f
T6:   a
T7:   b
T8:   c
T9:   x
T10:  y
T11:  z
T12:  *
```

השלימו את המשוואות הנוצרות מסוגי המבנים התחביריים הבאים (ב-ד), וציינו את הטיפוסים שניתן להסיק רק על פי המשוואות של מבנים אלו:

א. ביטויים פרימיטיביים

משוואות:

```
T12 = [number * number -> number]
```

טיפוסים מוסקים: לא ניתן להסיק דבר נוסף מעבר למה שמצויין במשוואות, כלומר רק את T12.

ב. ביטויים פרימיטיביים + הפעלת פרוצדורה/אופרטור **(5 נק)**

משוואות: _____

טיפוסים מוסקים: _____

_____

_____

ג. הגדרת פרוצדורה + מבנה תנאי **(5 נק)**

משוואות: _____

טיפוסים מוסקים: _____

_____

_____

ד. ביטויים פרימיטיביים + הגדרת פרוצדורה + הפעלת פרוצדורה/אופרטור **(5 נק)**

משוואות: _____

טיפוסים מוסקים: _____

_____

_____

_____

שאלה 4 – CPS (26 נק)

א. מה ההבדל בין תהליך חישוב רקורסיבי לבין תהליך חישוב איטרטיבי? **(2 נק)**

_____

_____

_____

ב. נתון הממשק הפונקציונלי הבא עבור עצים בינאריים: **(10 נק)**

```
empty?
node->value
node->left-child
node->right-child
```

נתון עץ בינארי אשר המידע אשר בצמתיו יכול להכיל פרוצדורות (מובטח כי הפרוצדורות אינן מקבלות פרמטר).
כתוב פונקציה בסגנון CPS אשר סורקת את העץ בינארי ב-DFS באמצעות הממשק הנ"ל ואשר מחזירה רשימה של התוצאות של ההפעלות של הפרוצדורות אשר בצמתים. צמת אשר אינו מכיל פרוצדורה, לא משפיע על הרשימה.
דוגמה:

```
(define my-tree
  (list 22
        (list (lambda () 15)
              '()
              (list 24
                    (list (lambda () 16)
                          '()
                          '())
                    '()))
        (list #t
              '()
              '())))

(tree-app$ my-tree (λ (x) x))
→ '(15 16)
```

השלימו את הפונקציה ואת הגדרת הטיפוס שלה:

```
; Signature tree-app$(t, cont)
; Type: ____________________________________________________
(define tree-app$
  (λ (t cont)
    (cond [(empty? t) (cont t)]
          [(procedure? (node->value t))
     ____________________________________________
     ____________________________________________
     ____________________________________________
     ____________________________________________
          [else
     ____________________________________________
     ____________________________________________
     ____________________________________________
     ____________________________________________
```

ג. הציעו דרך לקרוא לפונקציית ה-CPS מהסעיף הקודם כך שיוחזר מספר הצמתים אשר מכילים פרוצדורות **(2 נק)**

```
(tree-apps$ my-tree
____________________________________________________)
```

ד. מימשו פרוצדורה בסגנון success-fail שמפעילה את ה-success continuation במידה שכל הצמתים בעץ הם פרוצדורות, ואת ה-fail continuation במקרה שאחד מהצמתים אינו פרוצדורה. על הפרוצדורה לעבור על הצמתים בסדר depth-first ולעצור את המעבר מיד אחרי שהצמת הראשון שאינו פרוצדורה נמצא. **(12 נק)**

```
; Purpose: determine whether all the nodes in t are procedures
; Signature tree-sf$(t, success, fail)

; Type: ____________________________________________________
(define tree-sf$
  (λ (t success fail)
    (cond [(empty? t) (success t)]
          [(procedure? (node->value t))
     ____________________________________________
     ____________________________________________
     ____________________________________________
     ____________________________________________
          [else
     ____________________________________________
     ____________________________________________
     ____________________________________________
```

שאלה 5 : תכנות לוגי (20 נק)

א. חשבו את ה-Unifiers הבאים: **(6 נק)**

```
unify[ t(X, f(a), X), t(g(U), U, W) ]
```

_____

_____

```
unify[ t(X, f(X), X), t(g(U), U, W) ]
```

_____

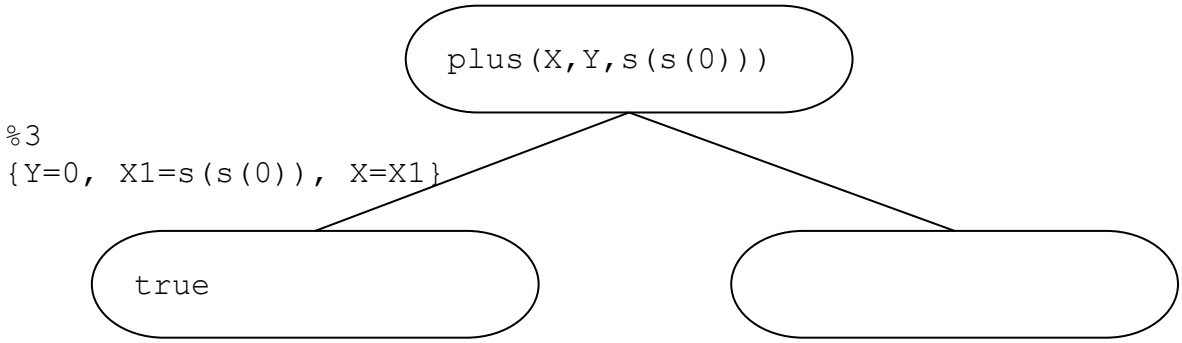
_____

ב. היזכרו בהגדרות עבור מספרי צ'רץ': **(6 נק)**

```
natural_number(0).                                    %1
natural_number(s(N)) :- natural_number(N).            %2

plus(X, 0, X).                                        %3
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).                 %4

times(_, 0, 0).                                       %5
times(X, s(Y), Z) :- times(X, Y, XY), plus(X, XY, Z). %6
```

השלימו את עץ ההוכחה עבור השאילתא (התרשים בעמוד הבא)

```
plus(X, Y, s(s(0))).
```


plus(X,Y,s(s(0)))
%3
{Y=0, X1=s(s(0)), X=X1}
true

ג. האם עץ ההוכחה מהסעיף הקודם הוא עץ הצלחה או כישלון? נמקו **(2 נק)**

_____

_____

_____

_____

ד. באמצעות מספרי צ'רץ' נגדיר מספרים רציונליים באופן הבא: **(6 נק)**

```
rational_number(q(X,Y)) :-
     natural_number(X),
     natural_number(Y),
     Y \= 0.
```

הגדירו את פעולת החיבור עבור מספרים רציונליים:

```
plus_r(q(X1,Y1), q(X2,Y2), q(X3,Y3)) :-
```

_____

_____

_____

_____