תאריך הבוחן: 10.5.2018
שם המרצה: מני אדלר,מיכאל אלחדד, ירון גונן
מבחן בקורס: עקרונות שפות תכנות
מס' קורס: 202-1-2051
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה
שנה: ב' סמסטר: ב'
משך הבוחן: 2 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 [30 נק'] AST

ניתנה הגדרת התחביר של שפת L2 ב-BNF וב-AST:

```
<exp> ::= <define> | <cexp>
<define> ::= ( define <var-decl> <cexp> ) / DefExp(var:VarDecl, val:CExp)
<cexp> ::= <number>                      / NumExp(val:number)
        |  <boolean>                     / BoolExp(val:boolean)
        | <prim-op>                      / PrimOp(op:string)
        | <var-ref>                      / VarRef(var:string)
        | (lambda (<var-dec>*) <cexp>+) / ProcExp(args: VarDecl[], body)
        | (<cexp> <cexp>*)               / AppExp(rator:CExp, rands:CExp[]))
<prim-op>  ::= + | - | * | / | < | > | =
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
```

```
export type Exp = DefineExp | CExp;
export type CExp = NumExp | BoolExp | PrimOp | VarRef | ProcExp | AppExp;
export interface DefineExp {tag: "DefineExp"; var: VarDecl; val: CExp};
export interface NumExp {tag: "NumExp"; val: number};
export interface BoolExp {tag: "BoolExp"; val: boolean};
export interface PrimOp {tag: "PrimOp", op: string};
export interface VarRef {tag: "VarRef", var: string};
export interface VarDecl {tag: "VarDecl", var: string};
export interface ProcExp {tag: "ProcExp", args: VarDecl[]; body: CExp[]};
export interface AppExp {tag: "AppExp", rator: CExp, rands: CExp[]};
```

ברצוננו להרחיב את השפה L2 עם פונקציה חדשה בשם bound? שבודקת אם משתנה קשור לערך ומתנהגת לפי הדוגמא הבאה

```
(bound? x) ➔ #f
(define x 1)  ➔ void
(bound? x) ➔ #t
((lambda (x) (bound? x)) 1) ➔ #t
```

1.א [3 נק'] סוג הביטוי

האם ניתן להגדיר bound? כפונקציה פרימיטיבית חדשה או כביטוי מיוחד חדש (special form) - נמק

bound? חייבת להיות ביטוי מיוחד בגלל שהפרמטר של bound? לא מחושב. בדוגמא לעיל:

(bound? x) לו היינו מחשבים את x כרגיל לפונקציה פרימיטיבית, היינו מקבלים טעות במקום #f

טעות נפוצה: שאלה (האים X או Y) אינה שאלה כן\לא – תשובה מסוג "כן" לא מתאימה

1.ב [6 נק'] מה הערך הצפוי לביטוים הבאים: בהנתן שהאינטרפרטר ממומש לפי **מודל ההחלפה (substitution model)**:

```
(bound? +)          ➔ #t

(bound? #t)         ➔ #t

(bound? (+ 1 2))    ➔ #f or Error לפי המימוש
```

bound? מאופיין כחישוב על משתנים – משמע שמצפים פרמטרים מסוג VarRef. אך השאלה מבררת כיצד bound? צריך להתנהג על ביטויים שאינם משתנים. נקודת המפתח היא שניתן שהאינטרפרטר ממומש לפי מודל ההחלפה. במודל הזה, משתנים שקשורים לפרמטרים של פונקציה מוחלפים בביטויים שקשורים לערכים של הפרמטרים. למשל:

```
((lambda (x) (bound? x)) 1)
```

בביטוי הזה – כשמפעילים את ה-closure על הערך המחושב 1 – מחליפים כל מופע חופשי של המשתנה x בביטוי valueToLitExp(1) – ואז מחשבים את תוצאת ההחלפה. במקרה של bound? חייבים להתיחס למופע x בתוך הביטוי (bound? x) כמופע חופשי – ולהחליף אותו (אחרת אין זכר לעובדה ש-x היה קשור לפרמטר המקומי של ה-closure). לכן כל פעם שנחשב (bound? E) על ביטוי E שיכול להיות תוצאה של valueToLitExp() נחזיר #t – אם מחשבים (bound? V) כאשר V הוא VarRef – נבדוק את V בתוך הסביבה הנוכחית (כמו בדוגמאהראשונה לעיל).

בדוגמאות שלנו - + הוא פרמטר מסוג PrimOp שיכול להיות תוצאה של valueToLitExp- כמו בקריאה:

((lambda (x) (bound? x)) +)

אותו דבר לגבי #t.

בניגוד לשני המקרים – הביטוי (+ 1 2) הוא AppExp ולא יכול להיות תוצאה של valueToLitExp – לכן נוכל להחליט להחזיר #f במקרה הזה או להחזיר Error.

טעויות נפוצות:

- להתעלם מהעובדה שהדוגמא ((lambda (x) (bound? x)) 1) חייבת להחזיר #t לפי מודל ההחלפות (ולכן לתעון שכל קריאה של bound? על ביטוי שאינו VarRef מחזיר #f או Error)
- לחשוב שההחלטה ש-PrimOp מיוצג כ-VarRef או כ-PrimOp תלויה במודל החישוב (החלפות \ סביבות) – אין קשר בין 2 ההחלטות האלה
- לא להבין מה המשמעות של המילה "קשור" – שמייצגת יחס בין משתנה לערך. אין משמעות במילים כמו "הערך קשור לערך אחר" או "הביטוי #t קשור לערך" (רק מופעיפ של משתנים יכולים להיות קשורים או חופשיים).
  ראה
  https://www.cs.bgu.ac.il/~ppl182/wiki.files/class/notebook/2.4SyntacticOperations.html#Free-and-Bound-Variables
- חשוב לזכור שבמודל ההחלפות משתנים גלובליים (המוגדרים ע"י define) מיוצגים בסביבה – אך משתנים מקומיים (פרמטרים לפרוצדורות) אינם מיוצגים בסביבות. בניגוד לזה, במודל הסביבות כל המשתנים מיוצגים בצורה אחידה בסביבות.

ג.1 [7 נק'] הרחב את הBNF ואת הAST

הרחב את ה-BNF ואת ה-AST של L2 כדי לתמוך ב-bound? כנדרש:

```
<cexp> ::= <number>                  / NumExp(val:number)
         | <boolean>                 / BoolExp(val:boolean)
         | <var-ref>                 / VarRef(var:string)
         | <primOp>                  / PrimOp(op:string)
         | (lambda (<var>*) <cexp>*) / ProcExp(args:VarDecl[], body: CExp[])
         | (<cexp> <cexp>*)          / AppExp(rator:CExp, rands:CExp[])

         | (bound? <var-ref>)        / BoundExp(var: VarRef | NumExp | BoolExp
                                                | ProcExp | PrimOp)
```

```
<prim-op>  ::= + | - | * | / | < | > | =
```

```
export type CExp = NumExp | BoolExp | PrimOp | VarRef | ProcExp | AppExp
                 | BoundExp
```

NOTE: The concrete syntax supports only <var-ref>, while the abstract syntax supports any type that can be returned by valueToLitExp. (This was not expected as part of the answer.)

טעות נפוצה:

- **לא להגדיר BoundExp כסוג של CExp**
- **התקבלה כל תשובה שבה BoundExp() מכיל פרמטר אחד מסוג string או VarRef או Cexp.**

ד.1 [14 נק'] הרחב את L2eval

כדי לתמוך בחישוב bound? כנדרש:

```
export type Value = number | boolean | PrimOp | Error;

const applyEnv = (env: Env, v: string): Value =>
    isEmptyEnv(env) ? Error(`var not found ${v}`) :
    env.var === v ? env.val :
    applyEnv(env.nextEnv, v);

const L2eval = (exp: CExp | Error, env: Env): Value =>
    isError(exp)  ? exp :
    isNumExp(exp) ? exp.val :
    isBoolExp(exp) ? exp.val :
    isPrimOp(exp) ? exp :
    isVarRef(exp) ? applyEnv(env, exp.var) :
    isProcExp(exp) ? makeClosure(exp.args, exp.body) :
    isAppExp(exp) ? applyProc(L2eval(exp.rator, env),
                              map((r) => L2eval(r, env), exp.rands),
                              env) :
   // If a new computation rule is needed - add here

    IsBoundExp(exp) ? evalBoundExp(exp, env) :

    Error(`Bad L2 AST ${exp}`)

const applyProcedure = (proc: Value, args: Value[], env: Env): Value =>
    isError(proc) ? proc :
    !hasNoError(args) ? Error(`Bad argument`) :
    isPrimOp(proc) ? applyPrimitive(proc, args) :
    isClosure(proc) ? applyClosure(proc, args, env) :
    Error("Bad procedure");

const applyPrimitive = (proc: CExp, args: Value[]): Value =>
    ! isPrimOp(proc) ? Error("Not a primitive") :
    proc.op === "+" ? args[0] + args[1] :
    proc.op === "-" ? args[0] - args[1] :
    proc.op === "*" ? args[0] * args[1] :
    proc.op === "/" ? args[0] / args[1] :
    proc.op === ">" ? args[0] > args[1] :
```

```
    proc.op === "<" ? args[0] < args[1] :
    proc.op === "=" ? args[0] === args[1] :
    // If a new primitive is needed add here
    Error("Bad primitive op " + proc.op);


const evalBoundExp = (e: BoundExp, env: Env): Value =>
    isVarRef(e.var) ? ! isError(applyEnv(env, e.var)) :
    isAppExp(e.var) ? false :
    true;
```

טעויות נפוצות:

- קריאה רקורסיבית ל-eval על הפרמטר var – קריאה כזות מיותרת ויכולה להחזיר טעויות שלא קשורות לעובדה שמשתנה קשור
- כתיבה מחדש של applyEnv כדי לטפל בטעות: אין צורך כי בקוד של האינטרפרטרים לא זורקים Error אלא מחזירים ערך מסוג Error (ולכן גם אין צורך ב-try ו-catch)

ראה
https://www.cs.bgu.ac.il/~ppl182/wiki.files/class/notebook/2.6SubstitutionModel.html#Error-Handling

שאלה 2 [15 נק'] Higher-order Functions

הגדר את הפונקציה **some** בשפת L4:
Some מקבלת 2 פרמטרים – פונקציה שמחזירה ערך בולאני (predicate) ורשימה. היא מחזירה #t כאשר אחד מהאיברים ברשימה מקיים את הפרדיקת.

2.א [5 נק'] הגדר את החתימה ו-type של some

```
;; Purpose: #t if one of the elts in the list satisfies the predicate

;; Signature: some(pred, elements)_____________________________

;; Type: ([T1 -> Boolean] * List(T1) -> Boolean]
```

2.ב [5 נק'] תן 2 דוגמאות שימוש:

```
;; Examples:
;; 1. Show behavior of some on an empty list

;; (some (lambda (x) (> x 0)) '())_____________          ➔

;; #f
```

הערה: הערך המוחזר של some על רשימה ריקה נובע מהעובדה ש-#f היא ערך נוטראלי של אופרטור or – ראה דיון בפרמטר המאתחל של פונקציה reduce

```
;; 2. Show behavior on a non-empty list

;; (some (lambda (x) (> x 0)) '(-1 -2))          ➔

;; #f
```

2.ג [5 נק'] ממש את some

```
(define empty? (lambda (x) (eq? x '())))

(define some (lambda (pred elements)
    (if (empty? elements)
        #f
        (or (pred (car elements))
            (some pred (cdr elements))))))

;; Other solution: foldr is like reduce
(define some2 (lambda (pred elements)
    (foldr (lambda (elt acc) (or (pred elt) acc)) #f elements))

foldr is a function in Racket – but we discussed how to implement a close
variant called fold in L4 if needed (this was not required in the answer):

(define fold (lambda (f init elements)
  (if (empty? elements)
      init
      (fold f (f (car elements) init) (cdr elements)))))
```

שאלה 3 [15 נק'] Rename and Substitute

התבונן בהפעלה של ה-substitution (במובן המתמטי) הבאה:

הערה: ראה דוגמא ב:
https://www.cs.bgu.ac.il/~ppl182/Practice_Sessions/PPL182_PS5
והגדרות ב:

https://www.cs.bgu.ac.il/~ppl182/wiki.files/class/notebook/2.6SubstitutionModel.html#Renaming-and-Substitution-Operations

E o s = ((lambda (x)
(+ x 12
((lambda (y w) (+ y x w)) z)))) o {z = 24, w = 12}

I. Rename E:

Renamed E =
((lambda (x_1)
(+ x_1 12
((lambda (y_2 w_2) (+ y_2 x_1 w_2)) z))))

II. Rename s:

{z = 24, w = 12}

III. Substitute:

E o s =
((lambda (x_1)
(+ x_1 12
((lambda (y_2 w_2) (+ y_2 x_1 w_2)) 24))))

IV. Original expression:

כתוב ביטוי ב-L3 שהחישוב שלו במודל ההחלפות דורש את הפעלת ה-substitution E o s:

```
((lambda (z w)
   ;; this is E copied here
   ((lambda (x)
              (+ x 12
                 ((lambda (y w) (+ y x w)) z))))
 )
 24 12)
```

הערות: **כאשר מתיחסים ל-**substitution **כפעולה מתמטית, מבצעים** renaming **גם על** E **וגם על הביטוים בצד ימין של ההחלפה** s **(בניגוד למה שממומש בקוד של האינטרפרטר** L3-eval**).**
גם מתעלמים מה-body **של** E **– כי לא מחשבים אתו – רק מפעילים עליו החלפה**

שאלה 4 [15 נק'] disjoint union types

השלם את הקוד הבא ב-TypeScript לפי ה-pattern של ה- disjoint union types:

```
interface Point { tag: "Point"; x: number; y: number};
interface Circle { tag: "Circle"; center: Point; radius: number};
interface Square { tag: "Square"; upperLeft: Point; side: number};
type Shape = Circle | Square;

const isSquare = (x: any): x is Square => x.tag === "Square";

const isCircle = (x: any): x is Circle => x.tag === "Circle";

const isShape = (x: any): x is Shape => isSquare(x) || isCircle(x);

const makeSquare = (ul: Point, side: number): Square =>
    ({tag: "Square", upperLeft: ul, side: side});

const makeCircle = (center: Point, radius: number): Circle =>
    ({tag: "Circle", center: center, radius: radius});

// Compute the area of a geometric shape
// The Error in the return type is not necessary
const area(s: Shape): number | Error =>
    isCircle(s) ? Math.PI * s.radius * s.radius :
    isSquare(s) ? s.side * s.side :
    Error("Unknown shape");
```

שאלה 5 [10 נק'] lexical address

5.א השלם את שמות המשתנים בביטוי הבא בהתאם לכתובת הלקסיקלית שלהם:

```
(lambda (x y)
  ((lambda (x) ([+ free] [_x : 0 0] [ y : 1 1]))
   ([+ free] [ x : 0 0] [ x : 0 0])) 1 2)
```

5.ב השלם את הכתובות הלקסיקליות בביטוי הבא:

```
(lambda (a b c)
    (if ([eq? free] [b : 0 1] [c  0 : 2])
          ((lambda (c)
             ([cons free] [a : 1 0] [c : 0 0]))
           [a : 0 0])
          [b : 0 1]))
```

שאלה 6 [15 נק'] normal/applicative order

6.א רשום 3 סוגים של חישובים שגורמים לאסטרטגיות חישוב normal- ו-applicative להתנהג בצורות שונות:

- לולאה אין סופית כפרמטר לפונקציה שלא משתמשים בו
- ביטוי שמחזיר טעות כפרמטר לפונקציה שלא משתמשים בו
- ביטוי שמכיל side-effect

טעות נפוצה: אין הבדל בסמנטיקה של if-exp בין normal ו-applicative.

6.ב רשום יתרון אחד וחיסרון אחד של applicative יחסית ל-normal:

יתרון: **יעילות** – במידה ופרמטר לפרוצדורה מופיע יותר מפעם אחת ב-body – ובהפעלה של הפרוצדורה מעבירים ביטוי מורכב כארגומנט – הביטוי המורכב

יחשב רק פעם אחת.

חיסרון: **בטיחות** – באחד משלושת המקרים שרשומים כסעיף א' – קיים ערך לביטוי שאפשר לחשב, normal מחזיר את הערך האמיתי הזה – בלי להתקע בטעות או לולאה אין סופית – אך applicative לא מצליח להחזיר את הערך הזה.