מבחן בקורס: עקרונות שפות תכנות, 202-1-2051
מועד: ב'
תאריך: 22/7/2019
שמות המרצים: מני אדלר, בן אייל, גיל אינציגר, מיכאל אלחדד, ירון גונן
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה, שנה ב', סמסטר ב'
משך המבחן: 3 שעות
חומר עזר: אסור

הנחיות כלליות:

- מבחנים שיכתבו בעיפרון חלש, המקשה על הקריאה, לא יבדקו.
- יש לענות על כל השאלות בגיליון התשובות. מומלץ לא לחרוג מן המקום המוקצה.
- אם אינכם יודעים את התשובה, ניתן לכתוב 'לא יודע' ולקבל 20% מהניקוד על הסעיף/השאלה.

שאלה 1: תחביר וסמנטיקה _____ _____ 35 נק
שאלה 2: טיפוסים _____ 25 נק
שאלה 3: CPS ורשימות עצלות_____ 30 נק
שאלה 4: תכנות לוגי _____ 20 נק

סה"כ _____ 110 נק

בהצלחה!

שאלה 1: תחביר וסמנטיקה
[35 נקודות]

נרחיב את השפה L2 כך שתכלול מבנה של לולאות.
הצורה המיוחדת for כוללת את משתנה הלולאה, את הערך ההתחלתי ואת ערך הסיום של משתנה זה בלולאה, ואת הביטוי לחישוב. בכל לולאה מחושב הביטוי על פי ערכו הנוכחי של משתנה הלולאה. ערך ביטוי ה-for הוא ערך הביטוי בלולאה האחרונה.

לדוגמא: בחישוב הביטוי `(for n 1 3 (* n n))`, יחושב התת-ביטוי (n n *) שלוש פעמים: עבור n=1, עבור n=2, ועבור n=3.
הערך של כל הביטוי הוא החישוב (n n *) עבור n=3 (הלולאה האחרונה) - 9.

שימו לב, כי הביטוי של הערך ההתחלתי והערך הסופי בלולאה (1 3 בדוגמא שהובאה קודם) יכול להיות ביטוי מספרי אך גם ביטוי שערכו מספרי (כמו x y, כאשר הם הוגדרו קודם עם ערך מספרי) - למשל:

```
(for n x (* 2 x) (+ n x))
```

להלן התחביר הקונקרטי והאבסטרקטי, וכן ה-ADT, של המבנה החדש:

```
<program> ::= (L2 <exp>+) // program(exps:List(exp))
<exp> ::= <define-exp> | <cexp>
<define-exp> ::= (define <var-decl> <cexp>) // def-exp(var:var-decl,
val:cexp)
<cexp> ::= <num-exp> // num-exp(val:Number)
       | <bool-exp>  // bool-exp(val:Boolean)
       | <prim-op>   // prim-op(op:string)
       | <var-ref>   // var-ref(var:string)
       | (if <exp> <exp> <exp>) // if-exp(test,then,else)
       | (lambda (<var-decl>*) <cexp>+)
                        // proc-exp(params:List(var-decl), body:List(cexp))
       | (<cexp> <cexp>*) // app-exp(rator:cexp, rands:List(cexp))
       | (for <var-decl> <cexp> <cexp> <cexp>) // for-exp(var:var-decl,
                                                low:cexp, high:cexp, body:cexp)
<prim-op> ::= + | - | * | / | < | > | = | not
<num-exp> ::= a number token
<bool-exp> ::= #t | #f
<var-ref> ::= an identifier token
<var-decl> ::= an identifier token
```

```
export interface ForExp {
  tag: "ForExp"; var: VarDecl; low: CExp; high: CExp; body: CExp
};
```

```
export const makeForExp =
   (var_: VarDecl, low: CExp, high: CExp, body: CExp): ForExp =>
      ({tag: "ForExp", var: var_, low: low, high: high, body: body});

export const isForExp = (x: any): x is ForExp => x.tag === "ForExp";
```

1.1 [5 נקודות]

להלן המימוש של חישוב המבנה for באינטרפרטר:

```
const applicativeEval = (exp: CExp | Error, env: Env): Value | Error =>
   isError(exp) ? exp :
   isNumExp(exp) ? exp.val :
   isBoolExp(exp) ? exp.val :
   isPrimOp(exp) ? exp :
   isVarRef(exp) ? applyEnv(env, exp.var) :
   isForExp(exp) ? evalForExp(exp, env) :
   isProcExp(exp) ? makeClosure(exp.args, exp.body) :
   isAppExp(exp) ? applyProc(L2eval(exp.rator, env),
                        map((r) => L2eval(r, env), exp.rands), env) :
   Error(`Bad L2 AST ${exp}`)

const evalForExp = (exp: ForExp, env: Env): Value | Error => {
    const low = applicativeEval(exp.low, env);
    const high = applicativeEval(exp.high, env);
    for (let i = low; i < high; i++) {
        applicativeEval(exp.body, makeExtEnv([exp.var.var], [i], env));
    return applicativeEval(exp.body, makeExtEnv([exp.var.var], [high], env));
}
```

אחד הסטודנטים בקורס טען, כי בניגוד למימוש המוצע, אין צורך לחשב את ה body של ה for שוב ושוב עבור כל ערך של משתנה הלולאה, אלא מספיק לחשב אותו פעם אחת עבור הערך האחרון של משתנה הלולאה. הסיבה לכך: בשפה L2 אין side effects, ולכן רק החישוב האחרון של ה body משפיע על ערך ביטוי ה for.

על פניו הסטודנט צודק, אך קיימים מקרים שבהם חישוב ביטוי ה-for על פי הצעת הסטודנט (בו מחושב ה-body רק עבור הערך הגבוה ביותר של משתנה הלולאה) אינו שקול לחישוב הביטוי ע"פ המימוש שלנו (בו מחושב ה-body לכל ערך בתחום של משתנה הלולאה).
ציינו מקרה כזה, ע"י הגדרת ביטוי for בשפה L2 המורחבת.

1.2 [16 נקודות]

1.2.1

- מה זה syntactic abbreviation
- ציינו שתי דוגמאות ל syntactic abbreviation (אין צורך בקוד, רק לציין מה ממירים למה)

[5 נקודות]

ניתן להמיר מבנה של for לסדרה של אפליקציות.
לדוגמא: ניתן להמיר את הביטוי

(for n 1 2 (* n n))

לסדרת הביטויים

```
((lambda (n) (* n n)) 1)
((lambda (n) (* n n)) 2)
```

הפרוצדורה For2Apps מבצעת המרה זאת באופן אוטומטי:

```
const For2Apps = (exp: ForExp, env: Env): AppExp[] => {
    const ret: AppExp[] = [];
    const low = applicativeEval(exp.low, env);
    const high = applicativeEval(exp.high, env);
    for (let i = low; i <= high; i++) {
        ret.push(makeAppExp(makeProcExp([exp.var], [exp.body]),
                                [makeNumExp(i)]));
    }
    return ret;
}
```

1.2.2 הסבירו מדוע המרה זו <u>אינה</u> המרה <u>תחבירית</u> (חִשבו במה היא שונה מהדוגמאות שציינתם בסעיף הקודם) [5 נקודות]

1.2.3 שנו את ההגדרה של הצורה for בתחביר הקונקרטי והאבסטרקטי, כך שניתן יהיה לבצע את ההמרה ברמה התחבירית (ניתן לשנות/לצמצם/למקד את טיפוסי תתי הביטויים במבנה for)
[3 נקודות]

1.2.4 עדכנו את המימוש של For2Apps בהתאם לשינוי שהצעתם בסעיף הקודם, כך שההמרה תהיה תחבירית.
[3 נקודות]

1.3 [14 נקודות]

בכיתה הצגנו את המימוש הבא של הפרוצדורה fact, המקבלת מספר ומחזירה את ערך העצרת שלו:

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

1.3.1 מימוש זה עושה שימוש ברקורסיה שאינה רקורסיית זנב. מה החיסרון של גישה זו? [2 נקודות]

1.3.2 ממשו מחדש (בשפה L2 המורחבת, הכוללת for ,!set) את הפרוצדורה fact באופן איטרטיבי, בשני אופנים:

- ע"י המרת הרקורסיה לרקורסיית זנב בטכניקת CPS
  (fact$ בדף התשובות)
- ע"י שימוש בלולאת for ובהשמה !set, ללא כל קריאה רקורסיבית (fact-for-set בדף התשובות).

אין להגדיר פונקציות עזר.

כזכור, הצורה המיוחדת !set מאפשרת לבצע השמה, כלומר לשנות את הערך של משתנה קיים.
לדוגמא, הערך של חישוב שלושת הביטויים הבאים הוא 80 ולא 5:

```
(define x 5)
(set! x 80)
x
```

[10 נקודות]

1.3.3 ציינו חיסרון אחד עבור כל אחד משני המימושים בסעיף הקודם [2 נקודות]

שאלה 2: טיפוסים [25 נקודות]

type unifiers 2.1
[6 נקודות]

כתבו את ה-Unifier של זוגות ביטויי הטיפוס הבאים.
במידה ולא קיים unifier - הסבירו למה:

2.1.1

```
[S * [Number -> S1] -> S]
[Pair(T1) * [T1 -> T1]-> T2]
```

2.1.2

```
[S * [Number -> S] -> S]
[Pair(T1) * [T1 -> T1] -> T2]
```

2.1.3

```
[T1 * [T1 -> T2] -> N]
[[T3 -> T4] * [T5 -> Number] -> N]
```

Records 2.2
[19 נקודות]

שפת ה-types שהגדרנו בכיתה ל-L5 מוגדרת לפי ה-BNF הבא:

```
<texp>          ::= <atomic-te> | <composite-te> | <tvar>
<atomic-te>     ::= <num-te> | <bool-te> | <void-te>
<num-te>        ::= number   // num-te()
<bool-te>       ::= boolean  // bool-te()
<str-te>        ::= string   // str-te()
<void-te>       ::= void     // void-te()
<composite-te> ::= <proc-te> | <tuple-te>
<non-tuple-te> ::= <atomic-te> | <proc-te> | <tvar>
<proc-te>       ::= [ <tuple-te> -> <non-tuple-te> ]
                    // proc-te(param-tes: list(te), return-te: te)
<tuple-te>      ::= <non-empty-tuple-te> | <empty-te>
<non-empty-tuple-te> ::= ( <non-tuple-te> *)* <non-tuple-te>
                    // tuple-te(tes: list(te))
<empty-te>      ::= Empty
<tvar>          ::= a symbol starting with T // tvar(id: Symbol)
```

נגדיר סוג חדש של type מורכב בשם Record לפי ההגדרה הבאה:

```
<record-te> ::= ( record <field-type>+ )
             // record-te(fields: list(fieldType))
<field-type> ::= (<string> : <non-tuple-te>)
             // fieldType(fieldName: string, fieldType: te)
```

כמו כן מגדירים שני פרימיטיבים חדשים ב-L5 לתמוך ב-records:

```
<prim-op> ::= + | - | * | / | < | > | = | not |  eq?
      | cons | car | cdr | pair? | list? | number?
      | boolean? | symbol? | display | newline
      | make-record | get-record
```

לדוגמא:

```
(define (r1 : (record ("a" : string) ("b" : number)))
     (make-record ("a" "s1") ("b" 2)))
(define double-b (lambda ((r : (record ("b" : number)))) : number
    (* 2 (get-record r "b")))
(double-b r1) → 4
```

2.2.1 [6 נק]

נגדיר את ה-typing rules ל-record באותה צורה כמו interfaces ב-TypeScript

הגדירו את קבוצת הערכים המתוארת ע"י ביטויי הטיפוס הבאים בעזרת פעולות של תורת הקבוצות.

ניתן להתייחס לקבוצת ה-fieldNames בשם Keys ולקבוצות כל הערכים האפשריים כ-V.

קבוצת הערכים מסוג string היא String, ו-Number עבור number.

השתמשו בפעולות בין קבוצות הבאות:

A U B, A ∩ B, A x B, A - B (הפרש),

A* היא קבוצת כל הסדרות האפשריות הכוללות איברים מהקבוצה A. כלומר, היא כוללת איברים מסוג:

$(a_1, \ldots, a_n)$, $a_i \in A$, for any natural number $n \geq 0$

לדוגמא:

A = {a, b}

A* = { (), (a), (b), (a, a), (a, b), (b, a), (b, b), (a, a, a), (a, a, b), … }

A**B היא קבוצת כל הסדרות האפשריות הכוללות זוגות איברים מהקבוצות A ו B, כאשר האיבר מהקבוצה A ('המפתח') מופיע רק פעם אחת בכל סדרה. כלומר היא כוללת איברים מסוג:

$((a_1, b_1), (a_2, b_2), \ldots (a_n, b_n))$, $\{a_1, \ldots, a_n\} \subseteq A$, $b_i \in B$, for any natural number $n \geq 0$.

לדוגמא:

A = {a, b}, B = {1, 2}
A**B = { (),
((a, 1)), ((a, 2)), ((b, 1)), ((b, 2)),
((a, 1), (b, 1)), ((a, 1), (b, 2)), ((a, 2), (b, 1)), ((a, 2), (b, 2)) }

הגדירו את הקבוצות עבור ה-types הבאים:

```
(record ("a" : string) ("b" : number))
```

```
(record ("a" : (record ("x" : number)) ("c" : boolean))
```

Type Inference 2.2.2
[13 נק]

התבוננו בביטוי הבא ב-L5:

```
(lambda (r)
   (* 2 (get-record r "b")))
```

<u>תזכורת</u>: ה-typing rules עבור ביטויים מסוג lambda, define, והאופרטור הפרימיטיבי *:

Typing rule Number:
For every type environment _Tenv and number _n:
_Tenv |- (num_exp _n) : Number

Typing rule String:
For every type environment _Tenv and string _s:
_Tenv |- (str_exp _n) : String

Typing rule Variable:
For every type environment _Tenv and variable _v:
_Tenv |- (varref _v) : Tenv(_v)

Typing rule Procedure:

For every: type environment _Tenv,
variables _x1, ..., _xn, n >= 0
expressions _e1, ..., _em, m >= 1, and
type expressions _S1, ...,_Sn, _U1, ...,_Um :
If _Tenv o {_x1:_S1, ..., _xn:_Sn } |- _ei:_Ui for all i = 1..m ,
then _Tenv |- (lambda (_x1 ... _xn) _e1 ... _em) : [_S1 * ... * _Sn -> _Um]

Typing rule Primitive *:

For every type environment _Tenv:
_Tenv |- * : (Number * ... * Number -> Number)

הגדירו את ה-typing rules עבור ה-primitive get-record כדי להשלים את ה-type inference של הביטוי.

Typing rule Primitive get-record:

For every type environment _Tenv:
_Tenv |- get-record: _____

תארו את השלבים של ה-type inference של הביטוי ואת התוצאה הסופית:

```
(lambda (r : Tr) : Tret
      (* 2 (get-record r "b")))
```

כאשר ה-Tvars הצמודים לתת-ביטויים הם:

- `T1: (lambda (r : Tr) : Tproc (* 2 (get-record r "b")))`
- `T*: *`
- `T2: 2`
- `Tget: (get-record r "b")`
- `Tb: "b"`

Type equations:

```
Expression                        Equation
```

Solve the equations:

`Solution:`

```
Tret = ____________________________________
Tr = ______________________________________
```

שאלה 3: CPS ורשימות עצלות [30 נקודות]

3.1 [4 נקודות]

הציעו קוד ב-TypeScript הבודק האם node.js מממש רקורסיית זנב בצורה איטרטיבית. הסבירו כיצד הקוד בודק זאת, ולאיזה תוצאות לצפות אם קיים מימוש איטרטיבי של רקורסיית זנב או אם לא.

3.2 [10 נקודות]

ממשו ב-L5 את הפרוצדורה map2$ המקבלת פרוצדורת CPS של שני פרמטרים, שתי רשימות באותו אורך ו-continuation, ומחזירה רשימה חדשה שהאיבר ה-i בה הוא ההפעלה של הפרוצדורה על האיבר ה-i בשתי הרשימות. צירפו את החוזה (contract) המלא של map2$. לדוגמה:

```
(map2$ +$ '(1 2 3) '(4 5 6) (lambda (x) x)) ;; -> '(5 7 9)
```

כאשר הגדרת +$ היא:

```
(define +$
  (lambda (a b cont)
    (cont (+ a b))))
```

3.3 [8 נקודות]

נתונה הפרוצדורה append וגרסת ה-CPS שלה, append$:

```
(define append
  (lambda (lst1 lst2)
    (if (empty? lst1)
        lst2
        (cons (car lst1) (append (cdr lst1) lst2)))))

(define append$
  (lambda (lst1 lst2 cont)
    (if (empty? lst1)
        (cont lst2)
        (append$ (cdr lst1)
                 lst2
                 (lambda (res)
                   (cont (cons (car lst1) res)))))))
```

הוכיחו (באינדוקציה) כי שתי הפרוצדורות הן שקולות-CPS, כלומר, לכל שתי רשימות lst1, lst2 ולכל פרוצדורה cont מתקיים:

```
(append$ lst1 lst2 cont) = (cont (append lst1 lst2))
```

3.4 [8 נקודות]

ממשו את הרשימה העצלה fibs - רשימה עצלה של מספרי פיבונאצ'י. השתמשו בפונקציית הממשק של רשימות עצלות cons-lzl.

דוגמה:

```
(take fibs 10) ;; -> '(0 1 1 2 3 5 8 13 21 34)
```

שאלה 4: תכנות לוגי [20 נקודות]

מטרת שאלה זו היא להשתמש במימוש של מספרים טבעיים שניתן באמצעות מנגנון הרשימות של פרולוג.
לדוגמא: הרשימה [] מייצגת את המספר 0 וכן הרשימה [1,1,1] מייצגת את המספר 3. באופן כללי אורך הרשימה קובע את המספר אותו היא מייצגת, וכן כל אברי הרשימה חייבים להיות

א. נתון הפרדיקט natural/1 שמכיל את כל המספרים הטבעיים (בייצוג רשימות).

```
% Signature: natural(X)/1

% Purpose: X is a (list representation) of a number.

natural([]).
natural([X|Xs]) :- natural(Xs).
```

a. [2 נקודות] ציירו את עץ החישוב עבור השאילתה: ?- natural([1,1])
b. [נקודה] מה סוג העץ המתקבל? האם הוא עץ הצלחה או כישלון סופי או אין סופי?
c. [2 נקודות] האם היחס מכיל רק מספרים טבעיים בייצוג רשימה? אם כן הסבירו מדוע, אם לא תקנו את היחס.

ב. [5 נקודות] ממשו את היחס equal/2 שמקבל שני מספרים בייצוג רשימות ובודק האם הם שווים. יש לבדוק שהקלט הוא בייצוג רשימות.

```
% Signature: equal(X, Y)/2.

% Purpose: X =Y.
```

ג. להזכירם: סידרת פיבונצ'י מוגדרת באופן הבא:

F1 = 1, F2 = 1, Fn = Fn-1 + Fn-2

a. [5 נקודות] ממשו את היחס fibonacci(N,X)/1 שמקבל אמת אם ורק אם X הוא מספר פיבונאצ'י ה-N-י, בייצוג רשימות, כאשר הקלטים X ו-N הם מספרים בייצוג רשימות (יש לוודא שהם מספרים בייצוג רשימות). לדוגמא:

```
fibonacci([1,1,1],[1,1]).

fibonacci([1],[1]).
```

b. [5 נקודות] כתבו שאילתה שמחזירה את כל מספרי פיבונאצ'י במקומות הזוגיים לפי הסדר - ללא מספרם ברשימה (F2, F4, F6 וכו').