מבחן בקורס: עקרונות שפות תכנות, 202-1-2051
מועד: א
תאריך: 22/7/2024
שמות המרצים: מני אדלר, ירון גונן, יובל פינטר
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה
משך המבחן: שעתיים
חומר עזר: אסור

הנחיות כלליות:

- 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 _____ נק 35
שאלה 2: מערכת טיפוסים _____ נק 20
שאלה 3: תכנות פונקציונאלי, CPS, רשימות עצלות נק 30
שאלה 4: תכנות לוגי _____ נק 15

סה"כ _____ נק 100

בהצלחה!

שאלה 1: תחביר וסמנטיקה [35 נקודות]

בתרגיל 2 הוספנו כזכור מבנה של class לשפה L3:

```
<program> ::= (L31 <exp>+)              / Program(exps:List(exp))
<exp> ::= <define> | <cexp>             / DefExp | CExp
<define> ::= ( define <var> <cexp> )    / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                  / VarRef(var:string)
<cexp> ::= <number>                     / NumExp(val:number)
        |  <boolean>                    / BoolExp(val:boolean)
        |  <string>                     / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(args:VarDecl[],
        |                                /         body:CExp[]))
        |  ( class ( <var>+ ) ( <binding>+ ) )
                 / ClassExp(fields:VarDecl[], methods:Binding[]))
        |  ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,
        |                                      then: CExp,
        |                                      alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ ) /
LetExp(bindings:Binding[],
        |                                      body:CExp[]))
        |  ( quote <sexp> )                / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )              / AppExp(operator:CExp,
        |                                         operands:CExp[]))
<binding>  ::= ( <var> <cexp> )            / Binding(var:VarDecl,
        |                                          val:Cexp)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
                 | cons | car | cdr | list | pair? | list? | number?
                 | boolean? | symbol? | string?
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

באופן זה ניתן:

- ליצור מחלקה

```
(define pair
    (class (a b)
         ((first (lambda () a))
```

```
            (second (lambda () b))
            (sum (lambda () (+ a b)))
           )
     )
)

pair
→ Class
```

- ליצור אובייקט ממחלקה

```
(define p34 (pair 3 4))
p34
→ Object
```

- להפעיל מתודה של אובייקט

```
(p34 'first)
→ 3
(p34 'second)
→ 4
(p34 'sum)
→ 7
```

להלן עיקר המימוש של מבנה המחלקה (במודל ההצבה):

```
export type ClassExp = {tag: "ClassExp"; fields: VarDecl[]; methods: Binding[]; }
export type ClassValue = {
    tag: "Class";
    fields: string[];
    methodNames: string[];
    methodProcs : CExp[];
}
export type ObjectValue = {
    tag: "Object";
    methodNames: string[];
    methodProcs : Closure[];
}

const L3applicativeEval = (exp: CExp, env: Env): Result<Value> =>
    isNumExp(exp) ? makeOk(exp.val) :
    isBoolExp(exp) ? makeOk(exp.val) :
    isStrExp(exp) ? makeOk(exp.val) :
    isPrimOp(exp) ? makeOk(exp) :
```

```
    isVarRef(exp) ? applyEnv(env, exp.var) :
    isLitExp(exp) ? makeOk(exp.val) :
    isIfExp(exp) ? evalIf(exp, env) :
    isProcExp(exp) ? evalProc(exp, env) :
    isClassExp(exp) ? evalClass(exp) :
    isAppExp(exp) ? bind(L3applicativeEval(exp.rator, env), (rator: Value) =>
                        bind(mapResult(param =>
                            L3applicativeEval(param, env),
                              exp.rands),
                            (rands: Value[]) =>
                                L3applyProcedure(rator, rands, env))) :
    isLetExp(exp) ? makeFailure('"let" not supported (yet)') :

    makeFailure('Never');


const evalClass = (exp: ClassExp): Result<ClassValue> =>
    makeOk(makeClassValue(map(vd=>vd.var,exp.fields),
                          map(b=> b.var.var, exp.methods),
                          map(b=> b.val, exp.methods)));

const L3applyProcedure = (proc: Value, args: Value[], env: Env): Result<Value> =>
    isPrimOp(proc) ? applyPrimitive(proc, args) :
    isClosure(proc) ? applyClosure(proc, args, env) :
    isClassValue(proc) ? applyClass(proc,args,env) :
    isObjectValue(proc) ? applyObject(proc,args,env) :
    makeFailure(`Bad procedure ${format(proc)}`);

const applyClass = (cls: ClassValue, args: Value[], env: Env):
Result<ObjectValue> =>
    bind(mapResult(proc => L3applicativeEval(proc,env),cls.methodProcs),
        (methods: Value[]) =>
            allT(isClosure, methods) ?
                makeOk(makeObjectValue(cls.methodNames, map((method : Closure) =>
                   subFieldsIntoMethod(method, cls.fields, args), methods)))
                :
                makeFailure("Non-proc method"));
```

```
const applyObject = (object: ObjectValue, args: Value[], env: Env): Result<Value>
=> {
    if (args.length < 1)
       return makeFailure("Wrong number of parameters for method application");
     else {
            if (isSymbolSExp(args[0])) {
                const i = object.methodNames.indexOf(args[0].val);
                if (i >= 0) {
                    const proc : Closure = object.methodProcs[i];
                    if (proc.params.length != args.length - 1)
                        return makeFailure("Wrong number of parameters");
                    return applyClosure(proc,args.slice(1),env);
                } else return makeFailure(`Unrecognized method: ${args[0].val}`);
            } else
             return makeFailure("Wrong type of first argument for method");
        }
    }

const subFieldsIntoMethod = (method : Closure, fields : string[], args : Value[])
: Closure => {
    const proc : ProcExp = makeProcExp(method.params, method.body);
    const renamedProc : CExp = renameExps([proc])[0];
    const litArgs : CExp[] = map(valueToLitExp,args);
    const substituedProc : CExp = substitute([renamedProc], fields, litArgs)[0];
    return makeClosure(substituedProc.args, substituedProc.body) :
}
```

א. ציינו מתי מחושבים המתודות של המחלקה והשדות של המחלקה, ומתי מתבצעת ההצבה של השדות ב-body של המתודות.
[10 נקודות]

המתודות מחושבות בזמן יצירת האובייקט (כלומר בהפעלת המחלקה עם האופרנדים לשדות). אז מחושבים גם השדות, ומתבצעת הצבתם בגוף המתודות.

טעויות נפוצות:

- **את ה-VarDecl של השדות לא מחשבים! חילוץ המחרוזת של שם השדה ממבנה ה-VarDecl אינו חישוב/קריא לפונקציית eval**
- **לא מספיק לענות 'בפרוצדורה L3ApplicativeEval', את כל הביטויים בעולם מחשבים שם.**
- **חישוב מתודה אינו הפעלה שלה! מדובר ביצירת הקלוז'ר שלה.**
- **הצבת שדות בתוך קלוז'ר אינה חישוב הקלוז'ר.**

ב. עדכנו את הקוד כך שהמתודות יחושבו בזמן יצירת המחלקה.
[10 נקודות]

```
export type ClassValue = {
    tag: "Class";
    fields: string[];
    methodNames: string[];
    methodProcs : Closure[];
}

const evalClass = (exp: ClassExp, env: Env): Result<ClassValue> =>
    bind(mapResult(binding => L3applicativeEval(binding.val,env),exp.methods),
        (methods: Value[]) =>
            allT(isClosure, methods) ?
                makeOk(makeClassValue(map(vd=>vd.var,exp.fields),
                                      map(b=> b.var.var, exp.methods),
                                      methods));
                :
                makeFailure("Non-proc method"));
```

ג. עדכנו את הקוד כך שהשדות יוצבו ב-body של המתודות רק בזמן הפעלתן.
[10 נקודות]

```
export type ObjectValue = {
    tag: "Object";
    fieldNames: string[];
    fieldValues: Value[];
    methodNames: string[];
    methodProcs : Closure[];
}

const applyClass = (cls: ClassValue, args: Value[], env: Env):
Result<ObjectValue> =>
   makeOk(makeObjectValue(cls.fields, args, cls.methodNames, cls.methodProcs));

const applyObject = (object: ObjectValue, args: Value[], env: Env): Result<Value>
=> {
    if (args.length < 1)
```

```
    return makeFailure("Wrong number of parameters for method application");
  else {
        if (isSymbolSExp(args[0])) {
            const i = object.methodNames.indexOf(args[0].val);
            if (i >= 0) {
              const proc : Closure = subFieldsIntoMethod(
               object.methodProcs[i], object.fieldNames, object.fieldValues);
              if (proc.params.length != args.length - 1)
                 return makeFailure("Wrong number of parameters");
               return applyClosure(proc,args.slice(1),env);
            } else return makeFailure(`Unrecognized method: ${args[0].val}`);
        } else
         return makeFailure("Wrong type of first argument for method");
    }
}
```

ד. איזו מהגישות עדיפה: זו שבמימוש הנוכחי, או זו שמוצעת בסעיפים ב-ג? נמקו בקצרה.
[5 נקודות]

זו סוגיה דומה לסדר אפליקטיבי מול סדר נורמאלי:

- **חישוב המתודות**
  אם יש מופעים רבים למחלקה כדאי לחשב את המתודות מראש
  אם אין מופעים למחלקה אין טעם לחשב מראש את המתודות
- **הצבת שדות במתודה**
  אם מפעילים אותה כמה פעמים כדאי להציב מראש.
  אם לא מפעילים אותה כדאי לא להציב מראש

למי שלא ציין גם את החיסרון של השיטה הנבחרת ירדה נקודה אחת (עם ההערה 'מצד שני')

שאלה 2: טיפוסים [20 נקודות]

באלגוריתם היסק הטיפוסים שלמדנו בשיעור, השלב השלישי הוא יצירת משוואות טיפוס מתוך כללי היסק הנוגעים לביטויים ספציפיים ב-L5. כך למשל, מתוך כלל ההיסק עבור פרוצדורות בעלות פרמטר אחד לפחות נורה לאלגוריתם ליצור משוואה כך:

Given: (lambda (v1 ...vn) e1 ... em), construct the following equation:

T_{lambda} = [T_{v1} * ... * T_{vn} -> T_{em}].

כאשר T_{lambda} הוא משתנה הטיפוס עבור הביטוי (lambda (v1 ...vn) e1 ... em).

בתרגיל 3 הכנסנו לשפה, בין היתר, את הביטויים union (איחוד טיפוסים) ו-diff (הפרש בין טיפוסים), את הביטוי any המבטא את קבוצת כל הטיפוסים, ואת פרדיקט הטיפוס שעבור טיפוס T יסומן is? T. לפרדיקט הטיפוס מעמד מיוחד על-פני בוליאן רגיל והוא משמש את מערכת בדיקת הטיפוסים.

א. [5 נק'] כתבו את כלל ההיסק, בצורה של הכלל לעיל, עבור פרוצדורה שבודקת טיפוס של פרמטר.

Given: (lambda (v) : is? T1 e1 ... em), construct the following equations:

T_{v} = any

T_{lambda} = [any -> is? T1]

במידה וחסרה המשוואה הראשונה, יש להוריד נקודה אחת.

ב. [5 נק'] כלל היסק שמשתנה בעקבות הכנסת הביטויים החדשים הוא הכלל ל-if. הסבירו באיזה אופן הכלל משתנה והשלימו את המשוואה הנוצרת.

הכלל משתנה בכך שהוא מאפשר החזרה של שני סוגי ביטויים דרך הייצוג של Union.
אין כאן משהו שמשתנה בגלל האפשרות של בדיקת טיפוס, לתשובות כאלה יש להוריד ציון.

Given: (if test then alt), construct the equation:

T_{if} = Union(T_{then}, T_{alt})

בחזרה ל-L5 הרגילה. מכלל ההיסק עבור הפעלת פרוצדורה הורינו לאלגוריתם לייצר את המשוואה הבאה:

Given: (op a1 ... an), construct the following equation:

T_{op} = [T_{a1} * ... * T_{an} -> T_{app}]

ג. [5 נק'] כתבו את כלל יצירת המשוואה עבור אופרטור let.

Given: (let ((v1 c1) ... (vn cn)) e1 ... em), construct the equations:

T_{v1} = T_{c1}

...

T_{vn} = T_{cn}

$T_{let} = T_{em}$

ד. [5 נק'] כזכור, את צורת let ניתן להמיר באופן דטרמיניסטי לביטוי של הפעלת פרוצדורה. האם משוואות שנוצרות מהפעלת פרוצדורה מעניקות יותר מידע על הטיפוסים מאשר משוואות על ביטוי let שקול? נמקו בקצרה.

בהמרה להפעלה יש התייחסות לטיפוסי המשתנים הלוקאליים שהם כעת הפרמטרים של האופרטור, אבל למרות זאת זה לא יותר אינפורמטיבי, כי המידע על האופרטור לא משפיע על מה שמחוץ ל-let, וזה לא תורם להיסק ב-let עצמו (המסיק את טיפוסי המשתנים הלוקאליים שלהם על פי טיפוס הערך שהם מאותחלים אליו, ובהתאם לכך את טיפוס הביטוי האחרון בבודי שהוא טיפוס ה-let כולו).

שאלה 3: תכנות פונקציונאלי, CPS, רשימות עצלות [30 נקודות]

א. [6 נק׳] תנו יתרון אחד וחיסרון אחד של שימוש ב-CPS.

יתרון: כל הביטויים בתוכנית יהיו בעמדת זנב, כלומר מחסנית הקריאות תכיל רק פריים אחד, מה שמאפשר חישובים איטרטיביים וטיפול ב-exceptions.

יתרון: מאפשר מס׳ פונקציות המשך, מה שמאפשר למשל סימולציה של exceptions בשפות שלא תומכות בזה.

חיסרון: יצירת הרבה קלוז׳רים ותפיסת מקום רב ב-heap

טעויות נפוצות:

- **קיבלנו חלקית טענה של ״קוד מסורבל״**
- **קיבלנו חלקית טענת חיסרון שצריך להסב את כל הקוד ל-CPS. השאלה היא על השימוש עצמו.**
- **טענות על יעילות חישוב לא התקבלו כי אין חישוב נוסף שמתבצע ב-CPS.**
- **טענות ש-״קל להמיר ל-CPS״ לא התקבלו כי זה לא יתרון. השאלה היא מדוע בכלל להמיר?**
- **טיעון ש-״אי אפשר לדעת מה פונקציית ההמשך״ לא התקבל, כי פונ׳ ההמשך היא כבר לא באחריותנו, כמו שללא CPS אני לא יכולים לדעת מי קורא לפונ׳ שלנו.**

ב. [6 נק׳] תנו שני מקרים בהם שימוש ברשימות עצלות הוא עדיף והסבירו מדוע.

- **עבודה עם רשימות אינסופיות**
- **עבודה עם רשימות התופסות מקום רב בזיכרון**
- **במידה והטיפול ברשימה מתבצע רק אם מתקיים תנאי מסויים, והתנאי לא מתקיים, הרשימה לא תיווצר כלל ויחסכו זיכרון וזמן חישוב.**
- **במידה ויש חישוב שהוא מבוסס טור, אפשר לחשב את הדיוק כרצוננו.**

ג. [6 נק׳] תנו שני מקרים בהם שימוש ברשימות עצלות הוא פחות עדיף והסבירו מדוע.

- **רשימות שצריך לעבור עליהן יותר מפעם אחת: בשימוש ברשימות עצלות נצטרך לייצר את הרשימה כמספר הפעמים שנעבור עליה, מה שמוסיף חישובים.**
- **מקרים בהם צריך גישה אקראית לאיברים ברשימה: נצטרך לייצר את כל האיברים עד לאיבר שנצטרך, ויש פה חישובים מיותרים.**
- **במידה ומדובר ברשימה סופית ונרצה לדעת רק מה האורך שלה, נצטרך לייצר את כולה כדי לבדוק זאת.**
- **למרות שזה לא פונקציונאלי, קיבלנו גם טענה של שינוי (מוטציה) של איברים ברשימה.**

טעויות נפוצות:

- **טענות על ״שפות לא פונקציונליות״ לא התקבלו כי יתכן מנגנון זהה גם בשפות לא פונקציונליות.**
- **טענה על ״רשימה קצרה״ ללא הסבר נוסף לא התקבלה. אם הרשימה היא קצרה, אבל עדיין צריך לעבור עליה רק פעם אחת, עדיין אין שום יתרון לרשימה רגילה.**
- **טענה על חישוב מסובך לא התקבלה כי ייתכן שנצטרך חישוב מסובך גם עבור רשימות קצרות.**
- **טענה על אי-חוקיות ביצירת האיברים לא התקבלה כיוון שניתן לקרוא רשימה עצלה גם מקובץ או ממקור ברשת.**

ד. [12 נק'] ממשו את הפונקציה take-while$, אשר תיכתב בסגנון CPS, אשר מקבלת רשימה עצלה ופרדיקט, ומחזירה רשימה עצלה כל עוד האיברים מרשימת הקלט מקיימים את הפרדיקט.
במימוש זה אנו מניחים כי פרוצדורת התנאי pred היא פרימיטיבית.

```
;; Signature:
;;  [[T1 -> Boolean] * Lzl<T1> * [Lzl<T1> -> T2] -> T2]
;; Example:
;;  (take (take-while$ (λ (n) (< n 5)) (ints-from 0) (λ (x) x)) 10)
;;  => '(0 1 2 3 4))
(define take-while$
  (λ (pred lzl cont)
    (if (or (empty-lzl? lzl) (not (pred (head lzl))))
        (cont empty-lzl)
        (take-while$
         pred
         (tail lzl)
         (λ (tail)
           (cont (cons-lzl (head lzl) (λ () tail))))))))
```

טעויות נפוצות:

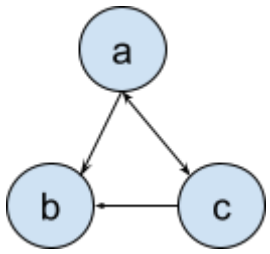
- **מימוש לא cps-י, על כל מה שמשתמע מכך – בנייה לא נכונה של cont מצעד לצעד; קריאה רקורסיבית במקום שאינו ה-tail position; הימנעות מהפעלת cont על רשימה עצלה ריקה בעת הגעה למקרי הבסיס, וכו'.**
- **מימוש filter ולא take-while - נדרשה פרוצדורה שתפסיק (ותפסיק לבנות את הרשימה העצלה) ברגע שהפרדיקט לא מתקיים, ולא שפשוט תדלג על האיבר שלא מקיים את הפרדיקט ותעבור לאיבר הבא (ברב המקרים פרוצדורה שכזו, לדוג', לא תפסיק לרוץ על רשימות עצלות אין סופיות).**
- **היעדר ציון אחד ממקרי הבסיס (כשמגיעים לרשימה ריקה, או כשהפרדיקט לא מתקיים), או שניהם.**
- **בנייה לא נכונה של lazy list (הכנסת האיבר השני באופן מפורש ולא באמצעות פרוצדורה חסרת פרמטרים).**
- **היעדר קריאות נוספות ל- take-while$, או קריאה ל-take-while$ במקום הלא נכון. בפרט, כאיבר הבא ברשימה עצלה שהתחילה להיבנות קודם. כלומר, הרשימה תתחיל להיבנות לפני שהגיעו לסוף הקריאות לפונקציה (ולמקרי הבסיס) – תקלה.**

שאלה 4: תכנות לוגי [15 נקודות]

ניתן לייצג עץ קשיר ומכוון בשתי דרכים:

- בעזרת הפרדיקט *edge/2* המציין צלע מכוונת בין שני קודקודים.
- על ידי רשימה של הצלעות

לדוגמא, ניתן לייצג את הגרף הבא בשתי דרכים:



- על ידי הפרדיקט edge/2

```
edge(a,b).
edge(a,c).
edge(c,b).
edge(c,a).
```

- על ידי רשימת הצלעות

```
[[a,b],[a,c],[c,b],[c,a]]
```

א. ממשו את הפרוצדורה pred_to_list/1 הממירה את הייצוג הראשון בשני.

לדוגמא, עבור העובדות הבאות:

```
edge(a,b).
edge(a,c).
edge(c,b).
edge(c,a).

?-pred_to_list(L)
L = [[a,b],[a,c],[c,b],[c,a]]

?-pred_to_list([[a,b],[a,c],[c,b],[c,a]])
true

?-pred_to_list([[a,b],[f,g]])
false
```

במימוש ניתן להשתמש בפרוצדורה findall/3 של פרולוג, המחזירה רשימה של נתונים מתוך העובדות בתוכנית, על פי תבנית נתונה. לפרוצדורה שלושה ארגומנטים:

1. תבנית המידע אותו רוצים להחזיר
2. תבנית העובדות מהן המידע נשלף
3. רשימה מוחזרת של מידע המתאים לתבנית העובדות כך שכל איבר ברשימה מתאים לתבנית המידע.

לדוגמא:

```
parent(abraham, issac).
parent(abraham, yishmael).

?-findall(Son,parent(abraham,Son),L)
L = [issac, yishmael]
```

```
pred_to_list(L):- findall([N1,N2],edge(N1,N2),L).
```

טעויות נפוצות:

- **קוד לא תקין תחבירית.**
- **שימוש לא נכון בארגומנטים של findall.**
- **ניתן ניקוד חלקי לפתרונות קרובים או נכונים חלקית (עם או בלי findall), למשל כאלה שמחזירים <u>גם</u> את הרשימה הרצויה `[[a,b],[a,c],[c,b],[c,a]]` אבל לא <u>רק</u> אותה.**
- **לא התקבל קוד שהועתק מסעיפים ב,ג.**

[5 נקודות]

ב. אחד הסטודנטים בקורס הציע להשתמש ב-not של פרולוג ולממש את pred_to_list באופן הבא:

```
pred_to_list(L):-pred_to_list([],L).        %1

pred_to_list(Acc,Acc):-                     %2
    edge(N1,N2),
    member([N1,N2],Acc).

pred_to_list(Acc,L):-                       %3
    edge(N1,N2),
    not(member([N1,N2],Acc)),
    pred_to_list([[N1,N2]|Acc],L).
```

הסבירו בקצרה מדוע ערך השאילתא הבאה הוא אמת (אין צורך לצייר את עץ ההוכחה):

```
edge(a,b).
edge(a,c).
```

```
edge(c,b).
edge(c,a).

?-pred_to_list([[a,b]])
true
```

[5 נקודות]

צריך להראות שמתקיים:

`pred_to_list([],[[a,b]])`

היוניפיקציה שלו מתאימה לחוק 3 עבור Acc=[]
עבור edge(a,b) צריך להתקיים

`pred_to_list([[N1,N2]|[]],[[a,b]])`

זה תקף על פי חוק 2.

הערה:

- **התקבלו גם תשובות שהסבירו במילים את מהות הקוד (L מכילה edges אבל לא בהכרח את <u>כולם</u>).**

ג. מה יהיה ערך השאילתא אם נהפוך את הסדר החוקים בפרוצדורה pred_to_list/2?

```
pred_to_list(L):-pred_to_list([],L).

pred_to_list(Acc,L):-
    edge(N1,N2),
    not(member([N1,N2],Acc)),
    pred_to_list([[N1,N2]|Acc],L).

pred_to_list(Acc,Acc):-
    edge(N1,N2),
    member([N1,N2],Acc).


?-pred_to_list([[a,b]])
```

[5 נקודות]

למדנו כי לכל שאילתא קיים עץ הוכחה יחיד עד כדי איזומורפיות. מאחר שהעץ כאן סופי, נגיע בהכרח לאותה תוצאה.