מבחן בקורס: עקרונות שפות תכנות, 202-1-2051
מועד: ב
תאריך: 12/8/2024
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]

א. כמה פריימים, מלבד הסביבה הגלובלית, נוצרים בביצוע תוכנית ב-L3, במודל ההצבה ובמודל הסביבות?
[5 נקודות]

מודל ההצבה: 0
מודל הסביבות: כמספר הפעלות הפונקציות

ב. כמה פריימים, מלבד הסביבה הגלובלית, נוצרים בביצוע תוכנית ב-L3 המורחבת, הכוללת גם את מבנה ה-class, במודל הסביבות?

מספר הפעלות הפונקציות והמתודות + מספר יצירת האובייקטים

בסס את תשובתכם על המימוש הבא של מבנה ה-class במודל הסביבות.

[5 נקודות]

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

const applicativeEval = (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) :
```

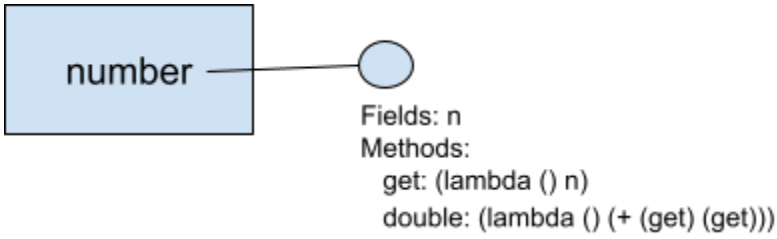
```
    isProcExp(exp) ? evalProc(exp, env) :
    isLetExp(exp) ? evalLet(exp, env) :
    isClassExp(exp) ? evalClass(exp,env) :
    isAppExp(exp) ? bind(applicativeEval(exp.rator, env),
                      (proc: Value) =>
                        bind(mapResult((rand: CExp) =>
                           applicativeEval(rand, env), exp.rands),
                              (args: Value[]) =>
                                 applyProcedure(proc, args))) :
    makeFailure('"let" not supported (yet)');

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

const applyClosure = (proc: Closure, args: Value[]): Result<Value> => {
    const vars = map((v: VarDecl) => v.var, proc.params);
    return evalSequence(proc.body, makeExtEnv(vars, args, proc.env));

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

const applyClass = (cls: ClassValue, args: Value[]): Result<ObjectValue> => {
        const extEnv = makeExtEnv(cls.fields, args,cls.env);
        return bind(mapResult((method: CExp) =>
            applicativeEval(method, extEnv), cls.methodProcs),
            (methods: Value[]) =>
                allT(isClosure, methods) ?
                   makeOk(makeObjectValue(cls.methodNames, methods, extEnv)) :
                    makeFailure("Non-proc method"));
}

const applyObject = (object: ObjectValue, args: Value[]): Result<Value> => {
    if (args.length < 1)
        return makeFailure("Wrong number of parameters");
```

```
        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));
                } else return makeFailure(`Unrecognized method: ${args[0].val}`);
            } else
                return makeFailure("Wrong type of first argument");
        }
    }
```

ג. נצייר את דיאגרמת הסביבות עבור מחלקות ואובייקטים באופן הבא:

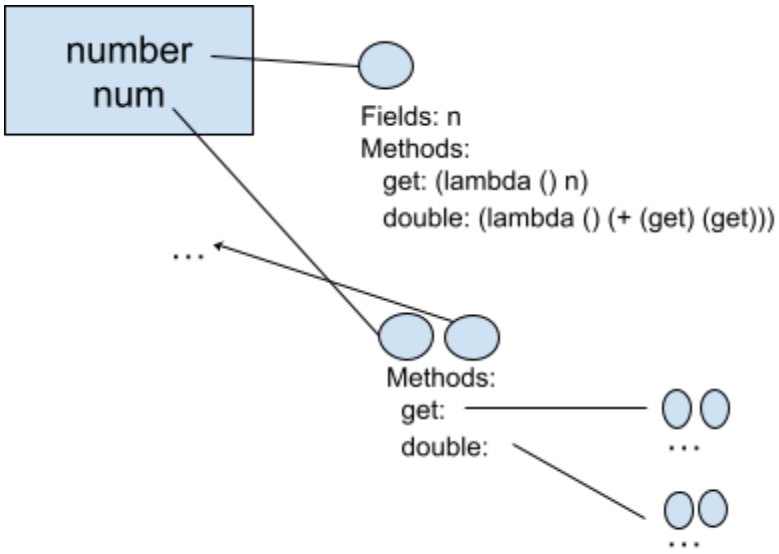
מחלקה:

```
(define number
    (class (n)
         ((get (lambda () n))
          (double (lambda () (+ (get) (get))))
         )
    )
)
```



אובייקט:

```
(define num (number 5))
```

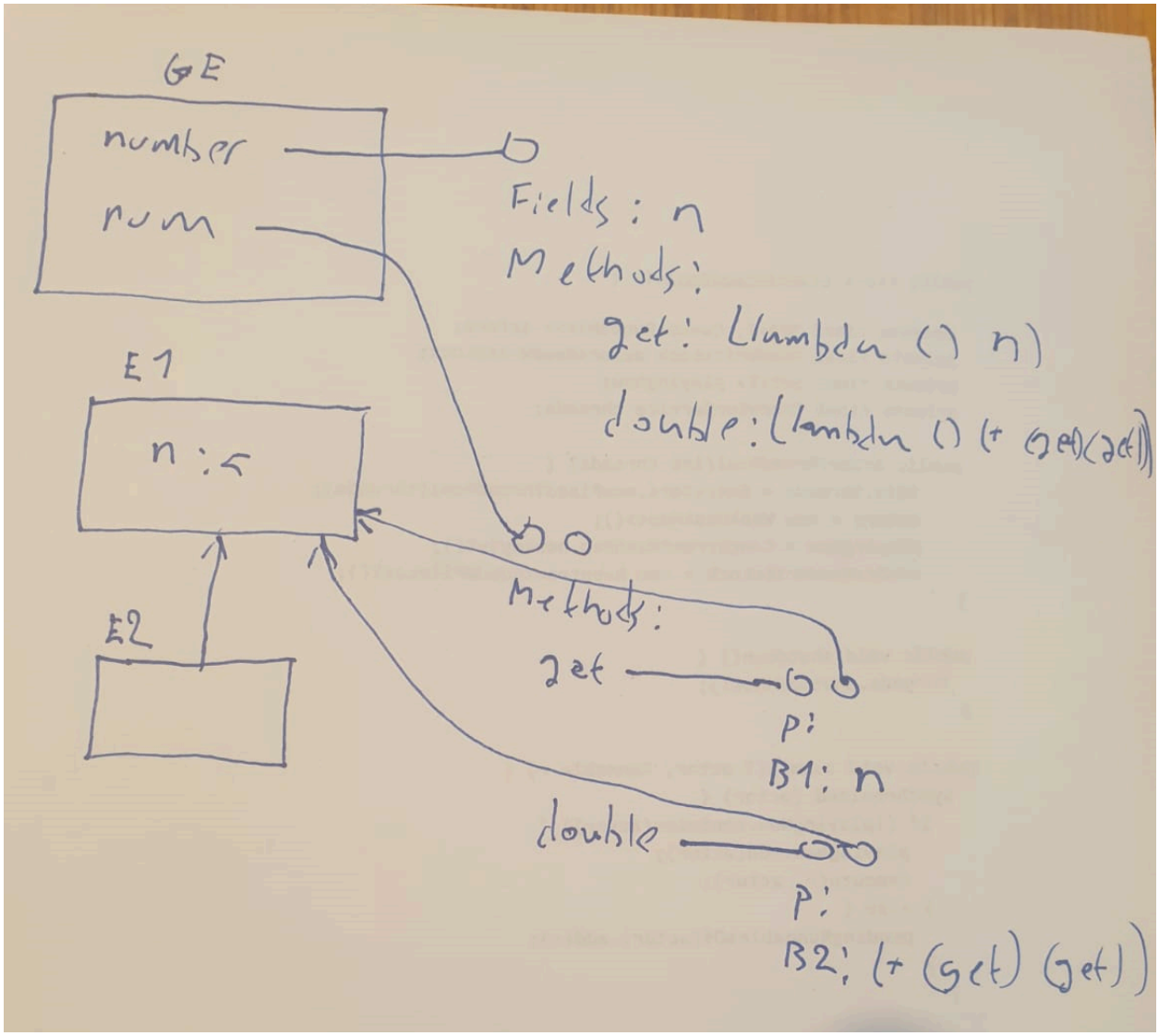


כאשר יש להחליף את שלוש הנקודות מתחת לקלוז'רים של המתודות get, double של number בתיאור הרגיל של קלוז'רים במודל הסביבות, ויש להחליף את שלושת הנקודות בסוף החץ שיוצא מהעיגול השני בתיאור האובייקט num לסביבה שבה הוא מוגדר.

השלימו את דיאגרמת הסביבות עבור חישוב התוכנית הבאה, והסבירו בעזרתה מה הבעיה בחישוב.

```
(define number
    (class (n)
         ((get (lambda () n))
          (double (lambda () (+ (get) (get))))
         )
    )
)
(define num (number 5))
(num 'double)
```

[10 נקודות]



double מפעילה את המתודה get אך זו לא מוגדרת בסביבה שלה E1, כי היא נוצרה על בסיס הסביבה שהיתה בזמן הפעלת ה-ClassValue.

ד. עדכנו את מימוש המתודה הבעייתית כך שהבעיה בסעיף ג לא תיווצר.

```
(define number
    (class (n)
         ((get (lambda () n))
          (double (lambda () (+ n n)))
         )
    )
)
```

[5 נקודות]

ה.

נתון כי בקובץ הסביבה של האינטרפרטר נוספה סביבה/פריים מסוג נוסף RecEnv:

```
export type Env = EmptyEnv | ExtEnv | RecEnv;
export type EmptyEnv = {tag: "EmptyEnv" }
export type ExtEnv = {
    tag: "ExtEnv";
    vars: string[];
    vals: Value[];
    nextEnv: Env;
}
export type RecEnv = {
    tag: "RecEnv";
    vars: string[];
    vals : ProcExp[];
    nextEnv: Env;
}

export const makeEmptyEnv = (): EmptyEnv => ({tag: "EmptyEnv"});
export const makeExtEnv = (vs: string[], vals: Value[], env: Env): ExtEnv =>
    ({tag: "ExtEnv", vars: vs, vals: vals, nextEnv: env});
export const makeRecEnv = (vs: string[], procs : ProcExp[], env: Env): RecEnv =>
  ({tag: "RecEnv", vars: vs, vals : procs, nextEnv: env});

export const applyEnv = (env: Env, v: string): Result<Value> =>
    isEmptyEnv(env) ? makeFailure(`var not found ${format(v)}`) :
    isExtEnv(env) ? applyExtEnv(env, v) :
    applyRecEnv(env, v);

const applyExtEnv = (env: ExtEnv, v: string): Result<Value> =>
    env.vars.includes(v) ? makeOk(env.vals[env.vars.indexOf(v)]) :
    applyEnv(env.nextEnv, v);

const applyRecEnv = (env: RecEnv, v: string): Result<Value> =>
    env.vars.includes(v) ? makeOk(makeClosure(env.vals[env.vars.indexOf(v)].args,
                                              env.vals[env.vars.indexOf(v)].body,
                                              env)) :
    applyEnv(env.nextEnv, v);
```

עדכנו את קוד האינטרפרטר שהופיע בסעיף ב כך שהבעיה בסעיף ג לא תיווצר.
[10 נקודות]

```
const applyClass = (cls: ClassValue, args: Value[]): Result<ObjectValue> => {
        const extEnv = makeRecEnv(cls.methodNames, cls.methodProcs,
                                        makeExtEnv(cls.fields, args, cls.env));
        return bind(mapResult((method: CExp) =>
            applicativeEval(method, extEnv), cls.methodProcs),
            (methods: Value[]) =>
                allT(isClosure, methods) ?
                   makeOk(makeObjectValue(cls.methodNames, methods, extEnv)) :
                    makeFailure("Non-proc method"));
}
```

שאלה 2: טיפוסים [20 נקודות]

א. [12 נק'] האם הצהרות הטיפוס הבאות נכונות או לא? אין צורך בהסבר.
ניתן להניח שבשפה הרלוונטית איחוד טיפוסים מוגדר, ומסומן בקו אנכי | .

1. `{f : [T2 → T3], g : [T1 → T4], a : Number} ⊢ (f (g a)) : T3`

   `True. T1 = Number, T4 = T2, and the resulting statement is correct.`

2. `{a: string, b: boolean, x: number} ⊢ (if (< x 0) a b) : string | boolean`

   `True.`

3. `{f : [number * [T5 → number] → T6], y : T5} ⊢ (f (g y)) : T6`

   `False. g remains free (= unknown, uninferrable) even after substitution. Also f is supposed to take two parameters.`

4. `{a: number, b: number} ⊢ (+ a b) : number`

   `True. + is primitive.`

ב. [8 נק'] בתרגיל 3 עסקנו בייצוג של איחודים וחיתוכים בין טיפוסים. ראינו בין היתר כי נדרש להמיר את תוצאת האיחוד או החיתוך של שני ביטויי טיפוס לצורה הקנונית DNF (איחוד של חיתוכים).
הוכיחו את הטענה הבאה, או הראו דוגמה נגדית: כל פעולת איחוד או חיתוך בין שני ביטויים הנתונים בצורת DNF ניתנת לחישוב והצגה כביטוי יחיד בצורת DNF ללא צורך בקריאה רקורסיבית. ניתן להניח כי מלבד איחודים וחיתוכים, השפה מורכבת מביטויי טיפוס אטומיים בלבד.

הוכחה: איחוד בין שני ביטויי DNF הוא פשוט איחוד הביטויים המאוחדים. הגדלת מספר הביטויים אינה מצריכה קריאה נוספת ל-DNF.
חיתוך בין שני ביטויי DNF ניתנים לפתיחה באמצעות חוק הפילוג: בלי הגבלת הכלליות (ולא באמת צריך פה שני איברים בכל חיתוך, זה בכלל לא משנה מה האיחוד מאחד כל עוד זה אפס או יותר נחתכים בכל איבר),
$((A \cap B) \cup (C \cap D)) \cap ((E \cap F) \cup (G \cap H)) =$
$(A \cap B \cap E \cap F) \cup (A \cap B \cap G \cap H) \cup (C \cap D \cap E \cap F) \cup (C \cap D \cap G \cap H) \leftarrow$ DNF

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

א. [6 נק׳] כתבו את הפונקציה deep-map ב-L4, אשר מקבלת פונקציה, ועץ, המיוצג על-ידי רשימה מקוננת של רשימות, ומפעילה את הפונקציה על כל איבר בעץ, תוך שמירה על מבנה העץ. הפונקציה מחזירה את העץ החדש:

```
;; Signature:
;;  [[T1 -> T2] * Tree<T1> -> Tree<T2>]
;; Example:
;; (deep-map add1 '((1 2) (3 (4 5) 6))) => '((2 3) (4 (5 6) 7))
(define deep-map
  (λ (f tree)
    (cond ((empty? tree) tree)
          ((list? tree) (cons
                         (deep-map f (car tree))
                         (deep-map f (cdr tree))))
          (else (f tree)))))
```

עוד פתרון יפה:

```
(define deep-map
  (λ (f tree)
    (map (λ(x) (if (list? x) (deep-map f x) (f x))) tree)))
```

טעויות נפוצות:

- התיחסות ל-(car tree) כאל איבר יחיד ולא תת עץ.

ב. [6 נק׳] ציינו שני יתרונות לכך שה-'משתנים' בתכנות פונקציונאלי הם immutable (כלומר לא ברי-שינוי):

1. קל יותר להוכיח נכונות של הקוד, כיוון שלא צריך להתייחס למקרים שערך משתנה.
2. קל יותר לכתוב אלגוריתם מקבילי כיוון שלא ייתכן race condition.
3. קל יותר לכתוב בדיקות לקוד כיוון שכל בדיקה אינה משנה את הסביבה לבדיקה אחרת.

טעויות נפוצות

- **טענות כמו ״אפשר לסמוך על התוכנית״ או ״לא צריך לחשב משתנים״ לא מתקבלות כיוון שהן ברמת הכותרת ולא נותנות יתרון ספציפי.**
- **הטענה ״אין חשיבות לסדר ביצוע התוכנית״ היא לא מדוייקת. יש חשיבות לביטוי האחרון בגוף של פונקציה למשל.**
- **טענה ״אין side effects״, בלי להסביר מה היתרון בזה, לא התקבלה.**
- **טענות על ״קוד יותר קריא״ לא התקבלו.**

- **טענות בעניין בדיקת טיפוסים לא התקבלו שכן בשפות שבהן יש mutations עדיין יש בדיקת טיפוסים. המשתנה מקבל ערך ערך אבל מאותו הטיפוס.**

ג. [6 נק׳] מה ההבדל בין תהליך החישוב של רקורסיית ראש ותהליך החישוב של רקורסיה הכתובה בסגנון CPS:
ברקורסיה רגילה התהליך החישובי גורר פתיחה של פריים על המחסנית בכל פעם שמתבצעת קריאה לפונקציה הרקורסיבית.
ברקורסיה בסגנון CPS הקריאה לפונקציה הרקורסיבית היא בעמדת זנב מה שהופך את תהליך החישוב לאיטרטיבי כמו חישוב של לולאה.

ד. [12 נק׳] ממשו את הפונקציה take-every-n-lzl המקבלת רשימה עצלה ומספר n, ומחזירה רשימה עצלה עם כל איבר n של הרשימה המקורית:

```
;; Signature:
;;  [Lzl<T1> * Number -> Lzl<T1>]
;; Example:
;;   (take (take-every-n-lzl (ints-from 1) 7) 2) => '(7 14)
(define take-every-n-lzl
  (λ (lzl n)
    (letrec ((loop (λ (lzl counter)
                     (cond ((= counter 1) (cons-lzl
                                           (head lzl)
                                           (λ () (loop (tail lzl) n))))
                           ((empty-lzl? lzl) lzl)
                           (else (loop (tail lzl) (- counter 1)))))))
      (loop lzl n))))
```

שאלה 4: תכנות לוגי [15 נקודות]

א. תארו את שלבי אלגוריתם היוניפיקציה עבור:

```
unify[ p([[v|V]|[V|VV]]), p([v|VV])]
```

[5 נקודות]

A = p([[v|V]|[V|VV]])
B = p([v|VV])

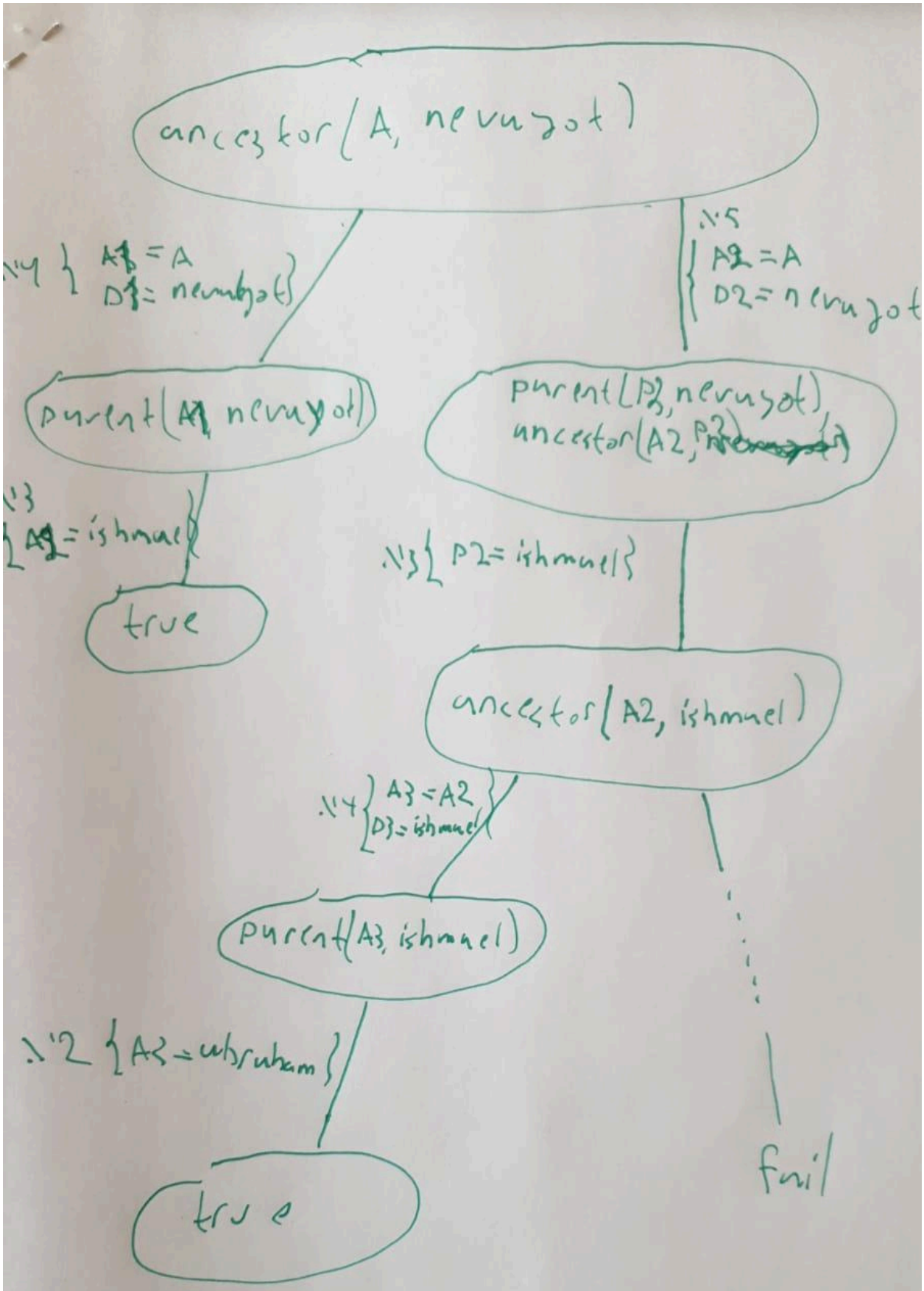
[v|V] = v - fail
Fails on constant equals list.

Another option:
[V|VV] = VV - fail
Fails on circularity.

ב. ציירו את מסלולי ההצלחה בעץ ההוכחה עבור השאילתא הבאה (אין צורך לצייר מסלולים שלא מובילים לפיתרון)

```
parent(abraham, isaac).        %1
parent(abraham, ishmael).      %2
parent(ishmael, nevayot).      %3

ancestor(A, D):- parent(A,D).   %4
ancestor(A, D):- parent(P, D), ancestor(A, P).    %5

?-ancestor(A, nevayot)
```

[10 נקודות]


ancestor(A, nevuzot)
N4 { A1 = A
D1 = nevuzot }
N5
A2 = A
D2 = nevuzot
parent(A1, nevuzot)
parent(P2, nevuzot)
ancestor(A2, P2)
N3
A1 = ishmael
true
N3 { P2 = ishmael }
ancestor(A2, ishmael)
N4 { A3 = A2
D3 = ishmael }
parent(A3, ishmael)
N2 { A3 = abraham }
true
fail