מבחן בקורס: עקרונות שפות תכנות, 202-1-2051
מועד: א
תאריך: 5/7/2022
שמות המרצים: מני אדלר, מיכאל אלחדד, ירון גונן
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה, שנה ב', סמסטר ב'
משך המבחן: 3 שעות
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 _____	35 נק
שאלה 2:	מערכת טיפוסים _____	20 נק
שאלה 3:	תכנות פונקציונאלי, CPS, רשימות עצלות	35 נק
שאלה 4:	תכנות לוגי _____	20 נק
סה"כ	_____	110 נק

בהצלחה!

שאלה 1: תחביר וסמנטיקה [35 נקודות]

א. הוספת הפרימיטיבים error, error?

בתרגיל 2 הגדרנו פונקציות משתמש לשם תמיכה בשגיאות.
כעת הוחלט להגדיר מנגנון זה, באופן מצומצם יותר, במסגרת השפה: האופרטורים הפרימיטיביים error?, make-error

```
(define div
  (lambda (x y)
    (if (= y 0)
      (make-error "div by 0")
      (/ x y))))
(define div1 (div 4 2))
(define div2 (div 4 0))
(error? div1)
→ false
(error? div2)
→ true
div1
→ 2
div2
→ <error: "div by 0">
```

א.1 הגדירו את הממשק Error המגדיר ערך חדש בשפה L3:

```
type Value = SExpValue;
type SExpValue = number | boolean | string | PrimOp | Closure |
SymbolSExp | EmptySExp | CompoundSExp | Error;

export interface Error {
   tag : "error";
   msg : string;
}

export const makeError = (msg: string): Error =>
   { tag : "error"; msg : msg; } ;

export const isError = (x : any) x is Error => x.tag === "Error" ;
```

הגדרת הטיפוס של tag כ string כללי, או של msg כטיפוס של מבנה תחבירי (SExp, StringExp) גררה הורדה של נקודה.

[6 נקודות]

א.2 הרחיבו את הפונקציה applyPrimitive כך שתתמוך בשתי הפעולות החדשות:

```
export const applyPrimitive = (proc: PrimOp, args: Value[]):
Result<Value> =>
    proc.op === "+" ? (allT(isNumber, args) ?
          makeOk(reduce((x, y) => x + y, 0, args)) :
          makeFailure("+ expects numbers only")) :
    proc.op === "-" ? minusPrim(args) :
    proc.op === "*" ? (allT(isNumber, args) ?
          makeOk(reduce((x, y) => x * y, 1, args)) :
          makeFailure("* expects numbers only")) :
    proc.op === "/" ? divPrim(args) :
    proc.op === ">" ? makeOk(args[0] > args[1]) :
    proc.op === "<" ? makeOk(args[0] < args[1]) :
    proc.op === "=" ? makeOk(args[0] === args[1]) :
    proc.op === "not" ? makeOk(!args[0]) :
    proc.op === "and" ? isBoolean(args[0]) && isBoolean(args[1]) ?
          makeOk(args[0] && args[1]) :
          makeFailure('Arguments to "and" not booleans') :
    proc.op === "or" ? isBoolean(args[0]) && isBoolean(args[1]) ?
          makeOk(args[0] || args[1]) :
          makeFailure('Arguments to "or" not booleans') :
    proc.op === "eq?" ? makeOk(eqPrim(args)) :
    proc.op === "string=?" ? makeOk(args[0] === args[1]) :
    proc.op === "number?" ? makeOk(typeof (args[0]) === 'number') :
    proc.op === "boolean?" ? makeOk(typeof (args[0]) === 'boolean') :
    proc.op === "string?" ? makeOk(isString(args[0])) :
    proc.op === "make-error" ? makeOk(makeError(args[0])) :
    proc.op === "error?" ? makeOk(isError(args[0])) :
    makeFailure("Bad primitive op " + proc.op);
```

אי שימוש בפונקציות שהוגדרו בסעיף א - isError, makeError - גררו הורדה של 2 נקודות.

[4 נקודות]

א.3 אחת הסטודנטיות בקורס הציעה לממש את make-error ו error? כצורות מיוחדות. האם הייתם מקבלים את הצעתה? נמקו בקצרה.
[3 נקודות]

צורות מיוחדות מוגדרות רק כאשר נדרש חישוב מיוחד שאינו תואם את ברירת המחדל של הפעלת אופרטור פרימיטיבי. כפי שראינו בסעיפים הקודמים, ניתן לממש את מנגנון ה-error על ידי אופרטורים פרימיטיביים, כך שאין צורך בחוקי חישוב מיוחדים וצורות מיוחדות חדשות.

א.4 כדי להתמודד עם אפשרות של הפעלת פונקציה עם ארגומנט שערכו הוא Error במסגרת האינטרפרטר, עדכנו את הפונקציה L3applyProcedure כך שבמידה ואחד הפרמטרים הינו Error, היא מחזירה שגיאה זו.

```
(+ 3 4)
→ 7
(+ (div 2 0) 4)
→  <Error: "div by 0">
(square 4)
→ 16
(square (div 2 0))
→ <Error: "div by 0">
```

```
const L3applyProcedure = (proc: Value, args: Value[], env:
Env): Result<Value> =>
    isError(proc) ? makeOk(proc) :
    !isEmpty(filter(isError, args)) ?
        makeOk(first(filter(isError, args))) :
    isPrimOp(proc) ? applyPrimitive(proc, args) :
    isClosure(proc) ? applyClosure(proc, args, env) :
    makeFailure("Bad procedure " + JSON.stringify(proc));
```

[4 נקודות]

טעויות נפוצות:

- אי התייחסות גם לפרוצדורה וגם לארגומנטים
- החזרה של Failure במקום OK (זו נקודה מהותית, מדובר בשגיאות שהן במסגרת השפה, חוזר ערך של Error, ולא שגיאות של האינטרפרטר)
- אי החזרה של אובייקט ה Error שהוגדר בסעיף א (החזרה של מחרוזת, הדפסה או כל קומבינה אחרת). גם זה עניין מהותי.

ב. מימוש normal order במודל הסביבות

ב.1 הראו דוגמת קוד שחישובה מסתיים ושעבורה applicative order יעיל יותר מ-normal order

(square (fact 100))

[2 נקודות]

ב.2 הראו דוגמת קוד שחישובה מסתיים ושעבורה normal order יעיל יותר מ-applicative order

((lambda (x y z) (if x y z)) #t 7 (fact 100))

[2 נקודות]

להלן שתי פונקציות מהאינטרפרטר של L4:

```
const eval = (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) :
    isLetrecExp(exp) ? evalLetrec(exp, env) :
    isAppExp(exp) ?
       bind(eval(exp.rator, env),
            (proc: Value) =>
               bind(mapResult((rand: CExp) =>
                eval(rand, env), exp.rands),
                (args: Value[]) =>
                   applyProcedure(proc, args))) :
    exp;

const applyProcedure = (proc: Value, args: Value[]): Result<Value> =>
    isPrimOp(proc) ? applyPrimitive(proc, args) :
    isClosure(proc) ? applyClosure(proc, args) :
    makeFailure(`Bad procedure ${JSON.stringify(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));
}

export const applyPrimitive = (proc: PrimOp, args: Value[]):
Result<Value> =>
// the implementation the function is not relevant for this question
…
```

ב.4 האם קוד זה מממש את מודל ההצבה או את מודל הסביבות? נמקו בקצרה [2 נקודות]

מודל הסביבות: מרחיבים את הסביבה ב applyClosure ולא משכתבים את ה body

ב.5 האם קוד זה מממש את applicative order או את normal order? נמקו בקצרה [2 נקודות]

applicative: מחשבים את האופרנדים לפני applyClosure

ב.6 כדי לממש normal order במודל הסביבות:

<u>הערה</u>: לשם פשטות, אופן המימוש שנדרש במבחן (המוצג בפתרון זה) אינו מכסה באופן מלא מקרים שבהם יש אופרנדים המוצבים בסביבה שונה מזו שבה הם הוגדרו.

- עדכנו את הטיפוס של השדה vals בממשק ExtEnv ובחתימת הפונקציות makeExtEnv בהתאם, ושנו בעקבות כך את applyEnv כך שחישוב המשתנים יהיה ב normal order:

```
export interface ExtEnv {
    tag: "ExtEnv";
    vars: string[];
    vals: ~~Value~~ CExp [];
    nextEnv: Env;
}

export const makeExtEnv = (vs: string[], vals: ~~Value~~ CExp[], env:
Env): ExtEnv =>
    ({tag: "ExtEnv", vars: vs, vals: vals, nextEnv: env});

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

- עדכנו את הפונקציה eval, ואת חתימות הפונקציות applyProcedure, applyClosure, applyPrimitive (שהופיעו למעלה) בהתאם.

```
const eval = (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) :
    isLetrecExp(exp) ? evalLetrec(exp, env) :
    isAppExp(exp) ?
       bind(eval(exp.rator, env),
            (proc: Value) =>
               ~~bind(mapResult((rand: CExp) =>~~
                ~~eval(rand, env), exp.rands)~~,
                ~~(args: Value[]) =>~~
                   applyProcedure(proc, exp.rands))) :
    exp;
```

```
const applyProcedure = (proc: Value, args: ~~Value~~ CExp[]): Result<Value> =>
    isPrimOp(proc) ? applyPrimitive(proc, args) :
    isClosure(proc) ? applyClosure(proc, args) :
    makeFailure(`Bad procedure ${JSON.stringify(proc)}`);

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

export const applyPrimitive = (proc: PrimOp, args: ~~Value~~ CExp[]):
Result<Value> =>
// the implementation the function is not relevant for this question
…
```

[10 נקודות]

שאלה 2: טיפוסים [20 נקודות]

בתרגיל 4, כזכור, נוספה לשפה L51 אפשרות של user-defined types, לפי ההגדרות הבאות:

```
<exp>  ::= <cexp> | <defineExp> | <defineTypeExp>
<cexp> ::= <atomicExp> | <procExp> | <litExp> | <ifExp> | <appExp> | <typecaseExp>

<defineTypeExp> ::= ( define-type <id> [( <id> <VarDecl>* )]* )
    / DefTypeExp(typeName:string, records:Record[])
    / Record(typeName:string, fields:VarDecl[])

<typecaseExp> ::= ( type-case <id> <CExp> ( <case-exp> )+  )
    / TypeCaseExp(typeName: string, val: CExp, cases: CaseExp[])

<case-exp> ::= (id (<varDecl>*) <cexp>+)
    / CaseExp(typeName: string, varDecls: VarDecl[], body: CExp[])
```

בשפה L51 נתן להגדיר טייפים חדשים בהתאם ל-disjoint union pattern:

```
(define-type Shape
    (circle (radius : number))
    (rectangle (width : number) (height : number)))

(define (area : (Shape -> number))
    (lambda ((s : Shape)) : number
        (type-case Shape s
            (circle (r) (* (* r r) 3.14))
            (rectangle (w h) (* w h)))))

(area (make-circle 1))
```

בשפה L51 קיימים יחסים של type/subtype בין ה-record types (לדוגמה circle ו-rectangle) וה-user defined types (בדוגמה Shape).

א. [10 נק'] בהינתן ההגדרות לעיל - עבור כל typing statement רשום האם ה-statement נכון - אם לא, הסבר למה:

- **{f:[Number->T1]} ⊢ (f 12): T1**

  True (because 12 is recognized as a number)

- **{x:circle, f:[Shape->T1]} ⊢ (f x):T1**

  True (because x belongs to circle, and circle is a subset of Shape, hence (f x) belongs to T1.

- **{x:Shape} ⊢ x: rectangle**

  False because rectangle is a subset of Shape, so if x belongs to Shape, it could also belong to circle and not belong to rectangle.

- **{x:circle} ⊢ x:Shape**

  True because circle is a subset of Shape.

- **{f:[T1->Shape], g:[circle->T2], x:T1} ⊢ (g (f x)):T2**

  False: (f x) is well typed because x belongs to T1 and f: T1->Shape. But (g (f x)) is not well typed because g expects a circle value, but (f x) could be a rectangle value.

ב. [5 נק'] בהשוואה בין מערכת הטייפים של Java ו-TypeScript, מבחינים בין:

structural subtyping (as in TypeScript)
nominal subtyping (as in Java)

התבוננו בשפה L51. האם L51 תומכת ב-**structural** או ב-subtyping **nominal**?
תנו דוגמה כדי לתמוך בתשובתכם.

The definition of structural vs. nominal is given in lecture notes in Type Checking | Principles of Programming Languages (bguppl.github.io)

- In structural subtyping (as in TypeScript) - the subtype relation among types is established by analyzing the members of the types.
- In nominal subtyping (as in Java and C++) - the subtype relation among types must be declared explicitly by the programmer.

L51 uses nominal subtyping. To demonstrate this - you must demonstrate a case where two records have similar fields (same types), but are not considered one a sub-type of the other.

For example, “rectangle” in the example above has type (width: number, height: number). If we added a record “square” as in:

```
(define-type Shape
    (circle (radius : number))
    (square (width : number))
    (rectangle (width : number) (height : number)))
```

Then, in L51 there is no way to derive that “rectangle” is a sub-type of “square” - while in a structural subtyping language, this relation would be inferred. (Indeed, this is not an intuitive relation - in mathematics, square is a subset of rectangle, but in the corresponding TypeScript type definition, rectangle would be a subtype of square because it extends its set of fields.)

The question focused on subtyping - other relations among types can be inferred in different languages and type systems. For example, in L51, we derive that “circle” and “rectangle” are disjoint. In a structural language like TypeScript, consider the declarations:

```
type circle = {radius: number}
type rectangle = {width: number, height: number}
```

They do not imply that the types are disjoint. We have to add a tag member to force them to become disjoint. In L51, the disjoint relation is implied by the define-type construct. This difference is also an indication that L51 is a nominal type system.

The subtype relation “rectangle < Shape” is not a good example to demonstrate that L51 is nominal. This relation is implied by the union construct. In TypeScript, if we define:

```
type circle = {radius: number}
type rectangle = {width: number, height: number}
type Shape = circle | rectangle
```

Then we infer that “circle < Shape” and “rectangle < Shape” - but this is a consequence of the semantics of union (component types of a union are subsets of the union), it does not indicate that TypeScript uses nominal subtyping.

ג. [5 נק'] האם ההגדרה הבאה של טייפ ב-L51 מגדירה טייפ חוקי - הסבירו:

```
(define-type T1
    (rec1 (f11 : number) (f12 : T2))
    (rec2 (f21 : string)))
(define-type T2
    (rec3 (f31 : T1))
    (rec4 (f41 : T2)))
```

A legal type must be one that defines possible finite values. A type that only contains infinite values would not be legal. For example:
`(define-type Bad (rec (f: Bad)))`
All possible values that belong to Bad would be infinite values of the form:
`(f (f (f ….)))`
Recursive types are legal in L51.
The question is whether this specific case of mutual recursion is legal - that is, can we find finite values that belong to T1 and to T2.

The answer is yes, T1 and T2 are legal types, because for each type there is a base case that can end the recursion of the values.
It is sufficient to show one finite value of T1 and one of T2 to establish this fact.
For example:

`(rec2 (f21 "a"))` **belongs to T1**

`(rec1 (f11 1) (f12 (rec3 (f31 (rec2 (f21 "a")))))))` **belongs to T1**

`(rec3 (f31 (rec2 (f21 "b"))))` **belongs to T2**

שאלה 3: תכנות פונקציונאלי, CPS, רשימות עצלות [35 נקודות]

א. [5 נק'] הסבירו מהי רקורסית זנב

פונקציה רקורסיבית כך שהקריאה הרקורסיבית נמצאת בעמדת זנב, כלומר היא הפעולה האחרונה בפונקציה. שימו לב כי זו תכונה *תחבירית*, כלומר להזכיר בסעיף זה אי פתיחת מסגרות על המחסנית זה לא נכון. עוד טעויות נפוצות: הגדרה של רקורסיה רגילה, פונ' שיוצרת חישוב איטרטיבי.

ב. [5 נק'] הסבירו מהי אופטימיזציה של רקורסיית זנב

מימוש של קריאות רקורסיביות כך שנוצר חישוב איטרטיבי ולא רקורסיבי.
המימוש מתבטא בכך שהקריאה הרקורסיבית אינה יוצרת מסגרת חדשה על מחסנית הקריאות. מימוש זה אפשרי רק במסגרת רקורסיית זנב, שכן במימוש כזה אין צורך לשמור את המסגרת הקודמת.
טעויות נפוצות: התהליך שהופך פונ' רקורסיבית ל-CPS.

ג. [5 נק'] הפונקציה remove-duplicates מקבלת רשימה lst, ומחזירה רשימה המכילה את כל האיברים מ-lst אבל ללא כפילות איברים. סדר האיברים ברשימת התוצאה הוא אותו הסדר כמו ב-lst. עבור איבר שמופיע יותר מפעם אחת, רק ההופעה הראשונה שלו נשמרת. דוגמאות:

```
(remove-duplicates '(1 1 1 1 1)) ⇒ '(1)
(remove-duplicates '(a b b a)) ⇒ '(a b)
(remove-duplicates '(1 2)) ⇒ '(1 2)
```

השלימו את קוד הפונקציה (רמז: השתמשו בפונקציה `filter`)

```
;; [List<T> -> List<T>]
(define remove-duplicates
  (λ (lst)
    (if (empty? lst)
        empty
        (cons
         (first lst)
         (remove-duplicates
          (filter (λ (x) (not (equal? (first lst) x)))
                  lst))))))
```

ד. [10 נק'] כתבו את הפונקציה בסגנון CPS - השלימו את החתימה. השתמשו ב-`filter$`.

```
;; Type [List<T1> * [List<T1> -> T2] -> T2]
(define remove-duplicates$
  (λ (lst cont)
    (if (empty? lst)
        (cont empty)
        (remove-duplicates$
         (cdr lst)
         (λ (res-cdr)
           (filter$
            (lambda (x cont) (cont (not (equal? (first lst) x))))
            res-cdr
            (λ (filter-lst)
              (cont (cons (car lst) filter-lst)))))))))
```

ה. [10 נק] היזכרו בפונקציה reduce:

```
;; Type: [[T1 * T2 -> T2] * T2 * List(T1) -> T2]
;; Purpose: Combine all the values of s using reducer
;; Example: (reduce + 0 '(1 2 3)) => (+ 1 (+ 2 (+ 3 0)))
(define reduce
  (lambda (reducer initial s)
    (if (empty? s)
        initial
        (reducer (car s)
                 (reduce reducer initial (cdr s))))))
```

כתבו את הפונקציה remove-duplicates תוך שימוש ב-reduce, **ללא קריאה רקורסיבית**.

```
(define remove-duplicates
  (λ (l)
    (reduce
     (λ (x y) (cons x (filter (λ (z) (not (eq? x z))) y)))
     empty
     l)))
```

טעויות נפוצות: פונקציות שבודקות האם פילטר מחזיר רשימה ריקה. זה קוד מייצג:

```
(define remove-duplicates
  (λ (l)
    (reduce
     (lambda (curr acc)
       (if (empty? (filter (λ (x) (eq? x curr)) acc))
           (cons curr acc)
           acc))
     '()
     l)))
```

או פונקציות שמשתמשות ב-member. קוד מייצג:

```
(define remove-dup-member
  (λ (l)
    (reduce
     (λ (curr acc)
       (if (member curr acc) acc
           (cons curr acc)))
     '()
     l)))
```

הפונקציות הללו יוצרות רשימות לא בסדר הנכון.

שאלה 4: תכנות לוגי [20 נקודות]

נתונים החוקים הלוגיים member, not_member
כזכור, X \= Y מצליח כאשר משתנה X אינו unifiable עם משתנה Y.

```
member(X, [X|_]).
member(X, [_|Ys]) :- member(X, Ys).

not_member(_, []).
not_member(X, [Y|Ys]):- X \= Y, not_member(X,Ys).
```

בשאלה זו נייצג קבוצה על ידי רשימה. כזכור, בקבוצה כל איבר מופיע פעם אחת.

א. ממשו את החוקים הלוגיים הבאים עבור קבוצות.
בחוקים intersection, union, disjoint, סדר האיברים בקבוצה השלישית נקבע על פי סדרם בשתי הקבוצות הראשונות.

```
% Signature: is_set(S)/1
% Purpose: check whether S is a set.
% ?- is_set([])
% true
% ?- is_set([1,2,3])
% true
% ?- is_set([1,2,1,3])
% false
is_set([]).
is_set([X|Xs]):- not_member(X,Xs), is_set(Xs).
```

<u>גרסה א</u>: המימוש מתבסס על ההנחה במבחן כי ניתן להניח סדר מסוים אחד של הקבוצה השלישית (סדר האיברים בשתי הקבוצות הראשונות במימוש זה):

```
% Signature: intersection(S1,S2,S3)/3
% Purpose: S3 is the intersection (חיתוך) of S1 and S2.
% ?-intersection([1,2,4],[2,3,1],[1,2])
% true
% ?-intersection([1,2],[3,4],[])
% true
% ?-intersection([1,1],[1],[1])
% false
intersection([],L,[]):-is_set(L).
intersection([X|Xs],L,[X|Ys]):-
```

```
   member(X,L),intersection(Xs,L,Ys),is_set([X|Xs]).
intersection([X|Xs],L,L2):-
   not_member(X,L),intersection(Xs,L,L2),is_set([X|Xs]).


Signature: union(S1,S2,S3)/3
% Purpose: S3 is the union (איחוד) of S1 and S2.
% ?-union([1,2],[3],[1,2,3])
% true
% ?-union([1,2],[3,3],[1,2,3])
% false
union([],L,L):-is_set(L).
union([X|Xs],L,[X|Ys]):-
  not_member(X,L), is_set([X|Xs]), union(Xs,L,Ys).
union([X|Xs],L,L2):-member(X,L), is_set([X|Xs]), union(Xs,L,L2).


% Signature: difference(S1,S2,S3)/3
% Purpose: S3 is the difference between S1 and S2 (S1 - S2).
% ?- difference([1,2,3],[1],[2,3])
% true
% ?- difference([1,2],[3,4],[1,2])
% true
% ?-difference([1],[1],[])
% true
% ?-difference([1,1,2],[2],[1])
% false
difference([],L,[]):-is_set(L).
difference([X|Xs],L,[X|Ys]):-
  not_member(X,L),difference(Xs,L,Ys),is_set([X|Xs]).
difference([X|Xs],L,L2):-
  member(X,L),difference(Xs,L,L2),is_set([X|Xs]).
```

<u>גרסה ב</u>: הרשימה השלישית ניתנת בכל סדר שהוא. לדוגמא עבור union:

```
union([], S2, S2).
union(S1, [], S1).
union([X1|X1s], S2, [X1|S3]) :- is_set([X1|X1s]), is_set(S2),
union(X1s, S2, S3), is_set([X1|S3]).
union([X1|X1s], S2, S3) :- is_set([X1|X1s]), is_set(S2), member(X1,
S2), union(X1s, S2, S3), is_set(S3).
union(S1, [Y2|Y2s], [Y2|S3]) :- is_set(S1), is_set([Y2|Y2s]),
not_member(Y2, S1), union(S1, Y2s, S3), is_set([Y2|S3]).
```

```
union(S1, [Y2|Y2s], S3) :- is_set(S1), is_set([Y2|Y2s]), member(Y2,
S1), union(S1, Y2s, S3), is_set(S3).
```

[16 נקודות]

ב. תנו דוגמאות לשאילתות על קבוצות (החוקים מסעיף א) שעץ ההוכחה שלהן הוא:

- עץ הצלחה סופי

```
?- difference([1,2,3],[1],[2,3])
```

- עץ כישלון סופי

```
?-difference([1,1,2],[2],[1])
```

- עץ הצלחה עם אינסוף תשובות

```
?-intersection(S1,S2,[])
```