**מבחן בקורס:** עקרונות שפות תכנות, 202-1-2051
**מועד:** א
**תאריך:** 27/6/2023
**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]

כדי לשחרר באופן יזום משתנים גלובליים שלא נדרשים יותר בתוכנית, הוצע להוסיף לשפה L4 (הממומשת עם box) צורה מיוחדת חדשה: undefine
undefine מקבל שם של משתנה, אם הוא מוגדר בסביבה הגלובלית הוא מוסר משם וחוזר true, אחרת חוזר false.
undefine הוא תמיד תת-ביטוי של program. לא ניתן להגדירו כתת-ביטוי של ביטויים אחרים.

```
(L4 (undefine x))
→ #f

(L4 (define x 7)
    (undefine x))
→ #t

(L4 (define x 7)
    x
    (undefine x)
    x)
→  { tag: 'Failure', message: 'Var not found: "x"' }
```

א. עדכנו את תחביר השפה L4 עם הצורה החדשה: התחביר הקונקרטי, התחביר המופשט, ומימוש המבנה התחבירי.
[8 נקודות]

**טעויות נפוצות:**

- **עדכון var בטיפוס string או VarDecl**
- **אי עדכון הכלל של exp או הוספת של undefine ל CExp או Program**
- **ערך החזרה של boolean ב isUndefExp**

```
<program> ::= (L4 <exp>+) / Program(exps:List(exp))
<exp> ::= <define> | <undefine> | <cexp>  / DefExp | CExp
<undefine> ::= (undefine <var>) / UndefExp(var:VarRef)
<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[])) |
  (if <cexp> <cexp> <cexp>) /
```

```
          IfExp(test: CExp, then: CExp, alt: CExp) |
  (let (<binding>*) <cexp>+) /
          LetExp(bindings:Binding[], body:CExp[])) |
  (letrec (binding*) <cexp>+) /
          LetrecExp(bindings:Bindings[], 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>* )

export type UndefExp = {
   tag : "undefine",
   var : VarRef

}

export const makeUndefExp = (var : VarRef): UndefExp =>
   { tag : "undefine", var : var };

export const isUndefExp = (x : any) : x is UndefExp => x.tag ===
"undefine";
```

ב. השלימו את מימוש הצורה החדשה באינטרפרטר.
[8 נקודות]

**טעויות נפוצות:**

- **שימוש לא נכון ב bind - אם יש Failure לא נגיע לחלק השני של bind ונחזיר failure במקום OK.**
- **החזרת boolean במקום Result<boolean>**
- **שימוש בפונקציה remove כללית ללא גישה ל fBindings**
- **הפעלה applicativeEval בתוך bind כדי לבדוק אם המשתנה מוגדר.**

```
const evalUnDefExp = (undef: UndefExp): Result<boolean> => {
```

```
   const old_bindings = unbox(theGlobalEnv.frame).fbindings;
   const new_bindings =
     old_bindings.filter(binding =>
                         binding.var.var !== undef.var.var);
    globalEnvSetFrame(theGlobalEnv,
     { tag: "Frame", fbindings: new_bindings});

    return makeOk(old_bindings.length != new_bindings.length);
}

const evalDefineExp = (def: DefineExp): Result<undefined> =>
    bind(applicativeEval(def.val, theGlobalEnv), (rhs: Value) =>
        {
            globalEnvAddBinding(def.var.var, rhs);
            return makeOk(undefined);
        });

type GlobalEnv = {
    tag: "GlobalEnv";
    frame: Box<Frame>;
}

export type Frame = {
    tag: "Frame";
    fbindings: FBinding[];
}

export type FBinding = {
    tag: "FBinding";
    var: string;
    val: Box<Value>;
}

const globalEnvSetFrame = (ge: GlobalEnv, f: Frame): void =>
                                  setBox(ge.frame, f);

export const globalEnvAddBinding = (v: string, val: Value): void =>
    globalEnvSetFrame(theGlobalEnv,
            extendFrame(unbox(theGlobalEnv.frame), v, val));

const applyGlobalEnvBdg = (ge: GlobalEnv, v: string):
Result<FBinding> =>
    applyFrame(unbox(ge.frame), v);
```

ג. האם ניתן לממש את undefine כפרוצדורת משתמש במקום צורה מיוחדת? נמקו בקצרה [1 נקודה]

**לא, אין גישה לסביבה מחוץ לאינטרפרטר**

ד. האם ניתן לממש את undefine כאופרטור פרימטיבי? נמקו בקצרה [4 נקודות]

**לא, באופרטור פרימיטיבי הפרמטר, כלומר ה-VarRef, יחושב לפני ההפעלה. כך שהאופרטור יקבל את ערכו ולא את ייצוגו כ-VarRef.**

**לאופרטור הפרימיטיבי יש גישה לסביבה הגלובלית, והוא יכול לבצע כל דבר שצורה מיוחדת יכולה לבצע - תשובות שהתבססו על כך הן שגויות.**

ה. הראו תרחיש שבו מבוצע '(undefine x)' בתוכנית, כאשר אחר כך אין שום התייחסות ל-x, ובכל זאת מתקבלת שגיאה (שלא היתה מתקבלת אם לא היינו מבצעים את undefine)
[5 נקודות]

```
(define x 7)
(define f (lambda () x)
(undefine x)
(f)
```

**תרחישים ללא קלוז'ר, כמו לדוגמא:**

```
(define x 7)
(define y x)
(undefine x)
 y
```

**אינן נכונים - בחוק החישוב של define מחושב ה-val קודם. כלומר מה שיש בסביבה אינו 'מצביע' ל-א אלא ערכו.**

ו. הציעו בקצרה דרך לפתור את הבעיה - יש לפרט (במילים) את השינויים שצריך לבצע באינטרפטר [6 נקודות]

- **נוסיף לטיפוס ה-FBinding שדה מספרי המציין את מספר הקלוז'רים שקשורים אליו (ref_count)**
- **בכל הגדרת קלוז'ר ב-define, נחלץ את רשימת המשתנים הגלובליים המופיעים ב-body שלו (מימשנו פונקציה כזאת באחת ההרצאות) ונגדיל את שדה ה-ref_count שלהם ב-1.**
- **כאשר מבצעים undefine למשתנה המייצג פונקציה והפעולה מצליחה, נקטין את ה-ref_count של המשתנים הגלובליים המופיעים ב-body שלו.**

- **כאשר מבצעים undefine למשתנה, נסיר אותו מהסביבה הגלובלית רק אם ה-ref_count שלו שווה ל-0.**

**אפשרויות אחרות:**

- **במימוש undefine בודקים האם המשתנה מופיע כ-VarRef ב-body של קלוז'רים.**

**אם כן:**

- **לא מבצעים את undefined ומחזירים false**

**או**

- **מציבים ב-body של הקלוז'ר את ערכו של ה-VarRef לפני שהוא נמחק**

ז. האם הבעיה שתארתם בסעיף ה תתרחש אם נממש את האינטרפרטר של L4 במודל ההצבה (עם box)? נמקו בקצרה.
[3 נקודות]

**כן. גם במודל ההצבה משתנים גלובליים מחושבים <u>בזמן הפעלת הפונקציה</u> (ולא בזמן הגדרתה, כפי שכתבו לא מעט סטודנטים) על פי הסביבה הגלובלית (בניגוד לפרמטרים של הפונקציה המוצבים בתוך ה-body)**

## שאלה 2: טיפוסים [20 נקודות]

2.1 בסעיף זה נתייחס למערכת הטיפוסים שמוגדרת בתרגיל 4 - L5 עם תוספת של union. כזכור union מוגדר כ-compound-TExp לפי התחביר:

```
UnionTExp ::= (union <TExp> <TExp>) / union-te(components: list(te))
```

לדוגמה:

`(union number boolean)` defines the type which contains all the number values and the boolean values.

`(union number (union string boolean))` defines the type which contains all the number values, string values and boolean values.

א. השלימו את ה-L5 type annotations של הביטוי הבא [2 נקודות]

```
(define f
  (lambda ((x : (union number boolean))) : (union boolean string)
    (if (not (boolean? x))
      (if (> x 0)
          "Number"
          #t)
      #f))))
```

**NOTES:**

1. **The syntax of L5/L51 expects the type of the return value of the function to appear after ":" after the parameters.**
2. **The type analysis of the body indicates that the return value would be (union (union string boolean) boolean) which is equivalent to (union boolean string)**
3. **The type analysis relies on the typing rule for IfExp defined in L51 which indicates that (if <test> <then> <else>) verifies that <test> has type boolean and returns (union typeOf(<then>) typeOf(<else>))**

ב. האם הביטויים הבאים type-safe? נמקו בקצרה.
[3 נקודות]

`(f (f #t))` **Yes - Type analysis of (f #t) shows this call returns #f - and then (f #f)**

| | |
|---|---|
| | **returns #f. The call is safe - it does not lead to a type error at runtime.** |
| `(f (f "a"))` | **No - The call (f "a") leads to the evaluation of (> "a" 0) which is a type error for the > primitive.** |
| `(f (if #t 1 #f))` | **Yes - the evaluation of (if #t 1 #f) yields 1 - and (f 1) returns "Number" in a safe way. The expression is type safe.** |

**NOTES:**

**An expression is type-safe if when we evaluate it we do not trigger a type error.** (See [Type Checking | Principles of Programming Languages (bguppl.github.io)](Type Checking | Principles of Programming Languages (bguppl.github.io)):

**programs … are type safe - if we compute them, on any possible input values, we do not reach type errors.**

**In the question, we had specific parameters passed to functions, so we can actually evaluate the expressions and verify whether they throw a type error.**

**This is not the same as determining that the Type Checker will accept these expressions and declare them as "type safe". This is "passing type checking" is not the same as "being type safe". This is the topic of the question as clarified in 2.2 - the type checker provides an approximation of type safety - if it declares an expression ok, then we can rely on it and be sure it will not throw a type error when evaluated (in other words, the type checker is sound), but if it declares an expression does not pass type checking - it does not entail the evaluation of the expression will necessarily lead to a type error (in other words, the type checker is not complete).**

**2.2**

נאמר כי type checker מקיים 'נְאוֹתוּת' (soundness) אם כאשר הוא קובע את הטיפוס של ביטוי נתון, קביעה זו תקפה לכל חישוב אפשרי.

א. האם ה-type checker של L5 מקיים נאותות?
[1 נקודה]

**Yes the type checker we developed is sound - we did not prove it in class, but the key properties of the type checking algorithm that achieve soundness are that it traverses the whole AST exhaustively and verifies every node in the expression, and that, through structural induction, it verifies the typing rules of every possible expression types. When we extended the algorithm to support Union types in L51 in Homework #4, we preserved**

**these properties and specified a systematic list of ways to compare any type expression as being a subtype of any other type expression.**

ב. ה-type checker אינו מקיים 'שלמות' (completeness), כי לעתים הוא מצביע על בעיית תאימות טיפוסים עבור ביטוי נתון, למרות שהביטוי עשוי להיות בטוח מבחינת הטיפוסים בזמן ריצה.

ציינו עבור שני המקרים הבאים האם יש בעיית חוסר שלמות ולמה:
[6 נקודות]

(f “a”)
**The type checker rejects this expression because “a” is not a member of (union number boolean.**
**At runtime, this expression leads to the evaluation of (> “a” 0) which is a type error.**
**Hence the type checker and the evaluation agree that this expression is not safe.**
**There is no case of lack of completeness.**

(f (f #t))
**The type checker analyzes that (f #t) is a safe call that returns a value in the type (union boolean string) (the return type of the function f).**
**Then the call (f (f #t)) is not accepted by the type checker because (union boolean string) is not a subtype of (union boolean number). Hence the type checker rejects the expression (f (f #t)) as unsafe.**
**The interpreter, however, succeeds to compute (f (f “t”)) as seen above and return #f without throwing a type error.**
**In this case, there is disagreement between the type checker analysis and the runtime behavior. It is a case of non-completeness of the type checker.**

## 2.3 Type Inference with Equations

בצעו את השלבים הראשונים של אלגוריתם type inference with type equations על הביטוי הבא:

```
(lambda ([f : Tf] [x : Tx]) : T1
  (lambda ([g : Tg]) : T2
    (f (+ x (g #t)))))
```

א. השמה של TVar לכל קודקוד ב-AST [שתי נקודות]

| Expression | Variable |
|---|---|
| (lambda (f x) (lambda (g) (f (+ x (g #t))))) | T0 |
| (lambda (g) (f (+ x (g #t)))) | T1 |
| (f (+ x (g #t))) | T2 |
| (+ x (g #t)) | T3 |
| (g #t) | T4 |
| f | Tf |
| x | Tx |
| g | Tg |
| + | T+ |
| #t | Tt |

ב. רשימת המשוואות [6 נקודות]

| Expression | Equation |
|---|---|
| (lambda (f x) (lambda (g) (f (+ x (g #t))))) | T0 = [Tf * Tx -> T1] (by ProcExp) |
| (lambda (g) (f (+ x (g #t)))) | T1 = [Tg -> T2] (by ProcExp) |
| (f (+ x (g #t))) | Tf = [T3 -> T2] (by AppExp) |
| (+ x (g #t)) | T+ = [Tx * T4 -> T3] (by AppExp) |
| (g #t) | Tg = [Tt -> T4] (by AppExp) |
| + | T+ = [Number * Number -> Number] (By PrimOp) |
| #t | Tt = Boolean (by LitExp typing rule) |

**Notes:**

1. **You must list the expression and the equation that derives from it**
2. **There are no equations derived from VarRef expressions**
3. **It helps to name type variables for variables with the variable names (as in Tx, Tf, Tg) to understand the equations.**
4. **For equations derived from AppExp, the left hand side is not the Tvar of the expression, but the TVar of the operator.**

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

3.1 מהי עמדת זנב (Tail Position)?
[2 נקודות]

**תכונה תחבירית וסמנטית של תת ביטוי כחלק מביטוי גדול יותר. התכונה אומרת שהערך של תת הביטוי הוא הערך של הביטוי כולו, כלומר תת הביטוי הוא החישוב האחרון שצריך לעשות כדי להעריך את הביטוי כולו.**
**תשובה מאוד יפה שראינו: "תת-ביטוי נמצא בעמדת זנב כאשר לאחר חישובו לא נדרש לבצע חישוב נוסף על מנת לקבל את הערך של הביטוי כולו".**
**טעויות נפוצות:**

- **"קריאות רקורסיביות", "פונקציה בעמדת זנב" - עמדת זנב היא תכונה של ביטוי שאינה קשורה בהכרח לרקורסיות או לפונקציות.**

3.2 מהו תהליך חישוב רקורסיבי?
[2 נקודות]

**תהליך חישוב של ביטוי אשר כולל בתוכו קריאה רקורסיבית, וחישוב הביטוי תלוי בתוצאות הקריאה הרקורסיבית, ולא יכול להסתיים עד שהקריאה הרקורסיבית מסתיימת.**

3.3 מהו תהליך חישוב איטרטיבי?
[2 נקודות]

**תהליך שבו חישוב של ביטוי כולל קריאה רקורסיבית, אבל חישוב הביטוי כולו הוא הערך של הקריאה הרקורסיבית, כלומר חישוב הביטוי לא צריך להמתין שהחישוב הרקורסיבי יסתיים.**

3.4 הפונקציה הבאה מוצאת את המספר הגדול ביותר מתוך רשימה של מספרים:
[2 נקודות]

```
;; Signature: max(lst)
;; Purpose: Finds the largest element of a list of numbers.
;; If the given list is empty it produces an error.
;; Type: [List -> Number union Void]
;; Example: (max '(2 9 5)) ⇒ 9
(define max
  (lambda (lst)
    (cond
      ((empty? lst) (error "Empty list!"))
      ((empty? (cdr lst)) (car lst))
      (else (let ((max-cdr (max (cdr lst)))
                  (first (car lst)))
              (if (> first max-cdr)
                  first
                  max-cdr))))))
```

מהו התת-ביטוי בפונקציה הגורם לתהליך החישובי להיות רקורסיבי ולא איטרטיבי? נמקו.

**הביטוי הוא: (((max-cdr (max (cdr lst))**

**מפני שהוא לא נמצא בעמדת זנב.**

3.5 השלימו את הקוד הבא, שהוא גירסה של הפונקציה max אשר מייצרת תהליך חישוב איטרטיבי:

[5 נקודות]

```
;; Signature: max-iter(lst)
;; Purpose: Finds the largest element of a number list.
;; If the given list is empty it produces an error.
;; The compuation is iterative.
;; Type: [List -> Number union Void]
;; Example: (max-iter '(2 9 5)) ⇒ 9
(define max-iter
  (lambda (lst)
    (letrec ((iter (lambda (lst max-so-far)
                     (if (empty? lst)
                         max-so-far
                         (if (> (car lst) max-so-far)
                             (iter (cdr lst) (car lst))
                             (iter (cdr lst) max-so-far))))))
      (if (empty? lst)
          (error "empty list")
          (iter (cdr lst) (car lst))))))
```

3.6 השלימו את הקוד הבא, שהוא גרסת CPS של max:

[5 נקודות]

```
;; Signature: max$(lst, succ-cont, fail-cont)
;; Purpose: Finds the largest element of a number list.
;; Type: [List * [Number -> T1] * [Empty -> Void] -> Number union
Void]
;; Example: (max$ '(2 9 5) id (lambda () (error "empty list"))) ⇒ 9
(define max$
  (lambda (lst succ-cont fail-cont)
    (cond ((empty? lst) (fail-cont))
          ((empty? (cdr lst)) (succ-cont (car lst)))
          (else
           (max$ (cdr lst)
```

```
(lambda (max-cdr)
  (if (> (car lst) max-cdr)
      (succ-cont (car lst))
      (succ-cont max-cdr)))
fail-cont)))))
```

3.7 כתבו ייתרון אחד שיש לגירסה האיטרטיבית על גרסת ה-CPS, והסבירו.

[3 נקודות]

**בגרסה האיטרטיבית גודל הזיכרון בערימה הוא קבוע. בגרסת ה-CPS ה-cont גדל כל הזמן.**

**טעויות נפוצות:**

- **"לא נפתחים פריימים" - בשתי הגרסאות לא נפתחים פריימים.**
- **"קוד יותר קריא וקל לתחזוק/להבנה" - זה עניין סוביקטיבי שלא ניתן למדידה.**

3.8 כתבו יתרון אחד שיש לגרסת ה-CPS על פני הגרסה האיטרטיבית, והסבירו.

[3 נקודות]

- **השימוש ב-cont עבור כישלון מאפשר גמישות בטיפול בכשלון, בעוד שבגרסה האיטרטיבית הטיפול בכישלון הוא קבוע.**
- **גרסאות איטרטיביות כוללות גם פרמטר "צבירה", מה שמשנה את הממשק באופן מהותי. שימוש ב-letrec כדי להתגבר על בעיה זו מצריך תמיכה של האינטרפרטר ב-letrec, מה שלא נדרש בגרסת ה-CPS.**
- **קל להוכיח שקילות לצורה המקורית בגלל שלא משנים את הממשק.**
- **אפשר להמיר אוטומטית פונקציות ל-CPS, מה שלא תמיד אפשרי באיטרציה.**

3.9 כתבו שני יתרונות לעבודה עם **רשימות עצלות** על פני רשימות רגילות.

[4 נקודות]

- **אפשרות לעבודה עם רשימות אינסופיות**
- **גודל הזיכרון הוא קבוע**

3.10 השלימו את הקוד עבור הפונקציה הבאה המקבלת גבולות של תחום, a ו-b, ומיצרת **רשימת עצלה** של כל המספרים השלמים בתחום זה, לא כולל הגבול העליון של התחום.

השתמשו בממשק לעבודה עם רשימות עצלות:

[7 נקודות]

```
head, tail, empty-lzl?, cons-lzl


;; Signature: range-lzl(a,b)
```

```
;; Purpose: Creates a finite lazy list containing all
;;          the integers between a and b, not including b
;; Type: [Number * Number -> Lzl]
(define range-lzl
  (λ (a b)
    (letrec ((loop (λ (n)
                     (if (= n b) empty-lzl
                         (cons-lzl n (λ () (loop (+ n 1)))))))))
      (loop a))))
```

**טעויות נפוצות: שימוש ב-'() כדי להציג רשימה עצלה ריקה, ולא שימוש נכון ב-API.**

## שאלה 4: תכנות לוגי [20 נקודות]

א. ממשו בשפה הלוגית את הפרוצדורה append2, הקובעת את היחס הבא: הפרמטר הראשון הוא רשימה של רשימות, והפרמטר השני הוא שרשור הרשימות האלה.
ניתן להניח שהפרמטר הראשון תמיד מוגדר בשאילתא (כלומר אינו משתנה)

```
% Signature: append2(Lists, List)/2
% Precondition: Lists is fully instantiated
%  (queries do not include variables in their first argument).


?- append2([ [a], [b] ], [a,b]).
true

?- append2([ [a], [b], [c] ], L).
L = [a,b,c]

?- append2([ [a], [[d,e]], [c] ], L).
L = [a, [d, e], c]
```

<u>אין</u> להגדיר חוקי עזר
<u>אין</u> להשתמש בפרוצדורות אחרות (בפרט לא ב-append ו-member שנלמדו בכיתה)

[10 נקודות]

```
append2([],[]).
append2([[]|Ls],L):-append2(Ls,L).
append2([[X|Xs]|Ls],[X|Ys]):-append2([Xs|Ls],Ys).
```

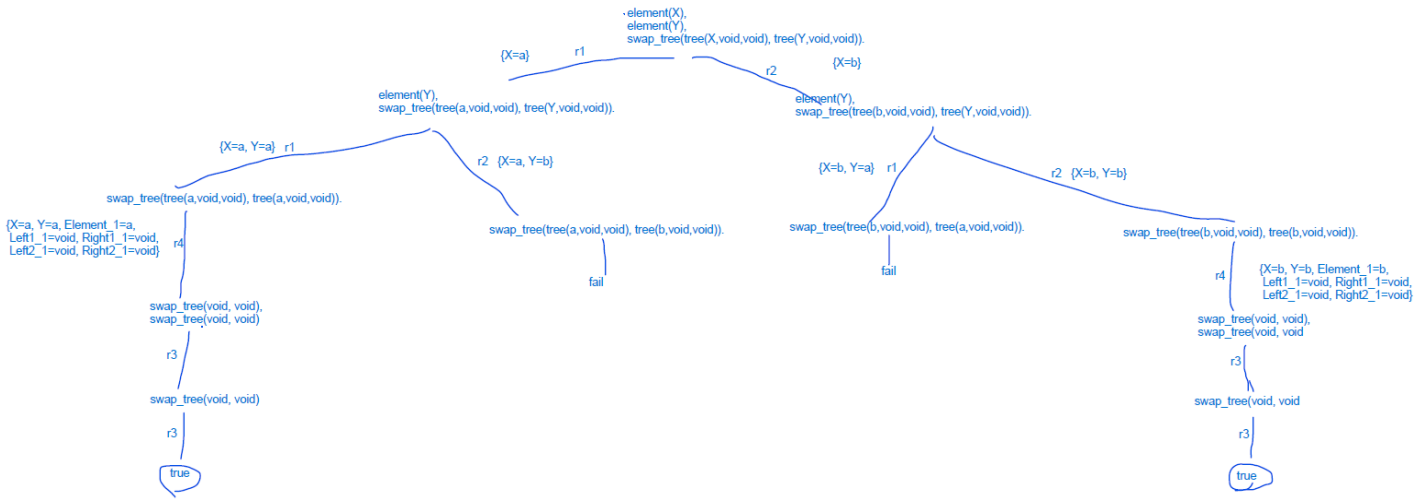
ב. נתונה התוכנית:

```
element(a).
element(b).
swap_tree(void,void).
swap_tree(tree(Element, Left1, Right1), tree(Element,Left2, Right2))
 :- swap_tree(Left1, Right2), swap_tree(Right1, Left2).
```

ציירו את עץ ההוכחה עבור השאילתא הבאה:

```
?- element(X), element(Y), swap_tree(tree(X,void,void),
tree(Y,void,void)).
```

[10 נקודות]


element(X),
element(Y),
swap_tree(tree(X,void,void), tree(Y,void,void)).
{X=a}
r1
r2
{X=b}
element(Y),
swap_tree(tree(a,void,void), tree(Y,void,void)).
element(Y),
swap_tree(tree(b,void,void), tree(Y,void,void)).
{X=a, Y=a}
r1
r2
{X=a, Y=b}
{X=b, Y=a}
r1
r2
{X=b, Y=b}
swap_tree(tree(a,void,void), tree(a,void,void)).
swap_tree(tree(a,void,void), tree(b,void,void)).
swap_tree(tree(b,void,void), tree(a,void,void)).
swap_tree(tree(b,void,void), tree(b,void,void)).
{X=a, Y=a, Element_1=a,
Left1_1=void, Right1_1=void,
Left2_1=void, Right2_1=void}
r4
fail
fail
r4
{X=b, Y=b, Element_1=b,
Left1_1=void, Right1_1=void,
Left2_1=void, Right2_1=void}
swap_tree(void, void),
swap_tree(void, void)
swap_tree(void, void),
swap_tree(void, void
r3
r3
swap_tree(void, void)
swap_tree(void, void
r3
r3
true
true