תאריך 02.07.2018
שם המרצים: מני אדלר, מיכאל אלחדד, ירון גונן
מבחן בקורס: עקרונות שפות תכנות
קורס' מס: 202-1-2051
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה
שנה: ב' סמסטר: ב'
משך הבוחן: 3 שעות
חומר עזר: אסור

הנחיות כלליות:

1. ההוראות במבחן מנוסחות בלשון זכר, אך מכוונות לנבחנים ולנבחנות כאחד.
2. מבחן הכתוב בעיפרון חלש המקשה על הקריאה, לא יבדק
3. יש לענות על כל השאלות בגוף המבחן בלבד (בתוך השאלון). מומלץ לא לחרוג מהמקום המוקצה.
4. אם אינך יודע את התשובה, ניתן לכתוב "לא יודע" ולקבל 20% מהניקוד על הסעיף/השאלה.

שאלה 1: lazy lists, AST _____ 25 נק
שאלה 2: ייצוג אובייקטים ב L, closures___ 20 נק
שאלה 3: הסק טיפוסים_____ 15 נק
שאלה 4: CPS _____ 26 נק
שאלה 5: תכנות לוגי _____ 20 נק
סה"כ_____ 106 נק

שאלה 1 - lazy lists, AST (25 נק)

א. ניתנה הגדרת ה-data type של lazy-lists כפי שהוגדרה בכיתה.
השלמו את ההגדרה של הפרוצדורה take - התייחסו למקרים שהרשימה קצרה מ-k איברים. **(3 נק)**

```
;; Lazy-list functional interface
(define cons-lzl cons)
(define empty-lzl empty)
(define head car)
(define tail
  (lambda (lzl)
    ((cdr lzl))))

;; Example lazy list
(define integers-from
  (lambda (n)
    (cons-lzl n (lambda () (integers-from (+ n 1))))))
(define integers (integers-from 0))

; Purpose: Return the first k elements of a lazy-list as a list
; Signature: take(lzl, k)
; Type: LZL(T) * Number -> List(T)_____________
(define take
  (lambda (lzl k)
    (if (or (eq? lzl empty-lzl) (= k 0))

        empty-lzl

        (cons (head lzl) (take (tail lzl) (- k 1))))))
```

Test 1: רשימה מספיק ארוכה
(take integers 3_____) =>

'(0 1 2)_____

Test 2: רשימה קצרה
(take empty-lzl 2_____) =>

'()_____

ב. הגדרו פרוצדורה **take-pad** שמקבלת lzl lazy-list, מספר k וערך כלשהו pad ומחזירה רשימה של בדיוק k איברים המכילה את האיברים הראשונים של lzl ובמידה ש-lzl קצרה את הערך pad עד לאורך k **(2 נק)**

```
(define short-lzl (cons-lzl 0 (lambda () empty-lzl)))
(take-pad short-lzl 3 '*)
→ '(0 * *)
(take-pad integers 3 '*)
→ '(0 1 2)

; Purpose: take exactly k elements from lzl with padding
; Signature: take-pad(lzl, k, pad)

; Type: LZL(T1) * Number * T2 -> List(T1 U T2)_______
(define take-pad
  (lambda (lzl k pad)
     (cond ((= k 0) empty)
           ((eq? lzl empty-lzl)
            (cons pad (take-pad lzl (- k 1) pad)))
           (else
            (cons (head lzl)
                  (take-pad (tail lzl) (- k 1) pad))))))
```

ג. הגדרו פרוצדורה **consecutive-pairs** שמקבלת lazy-list ומחזירה lazy-list של זוגות איברים רציפים מהקלט ללא חפיפה. במקרה שרשימת הקלט מסתיימת, השלמו את הזוג האחרון עם ערך מיוחד למילוי (padding). **(4 נק)**

```
(take(consecutive-pairs integers '*) 4)
→ '((0 1) (2 3) (4 5) (6 7))
(take (consecutive-pairs short-lzl '*) 4)
→ '((0 *))
```

```
; Purpose: compute lazy list of pairs of elements from lzl
; Signature: consecutive-pairs(lzl, padding)

; Type: [LZL(T1) * T2 -> LZL(List(T1 U T2))]
(define consecutive-pairs
  (lambda (lzl padding)
    (cond ((eq? lzl empty-lzl) empty-lzl)
          ((eq? (tail lzl) empty-lzl)
           (cons-lzl (cons (head lzl) padding)
                     (lambda () empty-lzl)))
          (else (cons-lzl
                  (cons (head lzl) (head (tail lzl)))
                  (lambda ()
                    (consecutive-pairs (tail (tail lzl))))))))
```

Can be shortened by reusing take-pad as:

```
(define consecutive-pairs
  (lambda (lzl padding)
    (cond ((eq? lzl empty-lzl) empty-lzl)
          ((eq? (tail lzl) empty-lzl)
           (cons-lzl (take-pad lzl 2 padding)
                     (lambda () empty-lzl)))
          (else (cons-lzl
                  (take-pad lzl 2 padding)
                  (lambda ()
                    (consecutive-pairs (tail (tail lzl)))))))))
```

ד. הגדר את הפרוצדורה ngram שמקבלת lazy-list ומחזירה רצפים של n איברים עם חפיפה.
במידה וה-lazy-list קצרה, כל האיברים של הרשימה חייבים להופיע בתוצאה - עם padding כנדרש. **(4 נק)**

```
(take (ngram integers 4 '*) 3)
→ '((0 1 2 3) (1 2 3 4) (2 3 4 5))
(take (ngram short-lzl 3 '*) 3)
→ '((0 * *))
```

```
; Purpose: Compute lazy list of ngrams from lzl
; Signature: ngram(lzl, n, pad)

; Type: LZL(T1) * Number * T2 -> LZL(List(T1 U T2))
(define ngram
  (lambda (lzl n pad)
    (if (eq? lzl empty-lzl)
        empty-lzl
        (cons-lzl (take-pad lzl n pad)
                  (lambda () (ngram (tail lzl) n pad))))))
```

ה. AST (12 נק)

ברצוננו להגדיר special-form חדש להקל על בניית lazy-lists. נגדיר את הביטוי החדש בשפה **make-lzl**:

```
(define lzl1 (make-lzl 1 empty-lzl))
;; equivalent to (cons-lzl 1 (lambda () empty-lzl))

(define lzl2 (make-lzl 2 lzl1))
;; equivalent to (cons-lzl 2 (lambda () lzl1))

(define integers (lambda (n) (make-lzl n (integers (+ n 1)))))
```

ניתן חלק מדקדוק השפה:

```
<cexp> ::= <number>        / NumExp(val:number)
  | <boolean>              / BoolExp(val:boolean)
  | <string>               / StrExp(val:string)
  | <prim-op>              / PrimOp(op:string)
  | <var-ref>              / VarRef(v:string)
  | (lambda (<var-decl>*) <TExp>* <cexp>+ ) /
        ProcExp(params:VarDecl[], body:CExp[], returnTE: TExp))
  | (if <cexp> <cexp> <cexp>) / IfExp(test: CExp, then: CExp, alt: CExp)
  | (<cexp> <cexp>*)          / AppExp(operator:CExp, operands:CExp[]))
```

השלמו את הדקדוק כדי לתמוך בביטוי מסוג **make-lzl** (BNF ו-abstract syntax) (3 נק)

```
  | (make-lzl <cexp> <cexp>+) / MakeLzlExp(head: Cexp, tail: Cexp[])
```

כמו כן הגדרו את ה-type הנדרש ב-AST: (3 נק)

```
interface MakeLzlExp {tag: "MakeLzlExp"; head: Cexp; tail: Cexp[] }

const makeMakeLzlExp = (head: Cexp, tail: Cexp[]): MakeLzlExp =>

    ({tag: "MakeLzlExp", head: head, tail: tail});

const isMakeLzlExp = (x: any): x is MakeLzlExp => x.tag === "MakeLzlExp";
```

השלם את הפרוצדורה שמשכתבת ביטוי מסוג **make-lzl** לביטוי אקוויוולנטי עם **cons-lzl** (6 נק)

```
// Purpose: rewrite a single makeLzl exp as a cons-lzl application
//          with nested lambda
// Signature: rewriteMakeLzl(makeLzlExp)

// Type: [MakeLzlExp -> AppExp]

const rewriteMakeLzl = (e: MakeLzlExp): AppExp =>
    MakeAppExp(MakeVarRef("cons-lzl"),
        [e.head,
         MakeProcExp([], e.tail)]);
```

NOTES:

Pay attention to these points:

- Material on the first sections is from https://www.cs.bgu.ac.il/~ppl182/wiki.files/class/notebook/4.3Generators.html#Manipulation-of-LZL-Values
- Always use accessors head and tail to take apart a lazy-list - using car / cdr / cadr will not work.
- Types: the first 2 functions (take and take-pad) take a LZL as argument and return a regular list. The next 2 functions (consecutive-pairs and ngram) take a LZL and return a LZL.
- Always use cons-lzl to construct a LZL with parameters of the form (cons-lzl <something> (lambda () <something which returns a list>))
- In consecutive-pairs, must test the case where LZL is empty and where (tail lzl) is empty. There must be a recursive call to consecutive-pairs.
- In ngram, reuse take-pad (don't copy it as a new letrec function), and make sure ngram is called recursively.
- One cannot use the function length on a lazy-list (because it could be infinite).

About the syntactic rewrite - material is from https://www.cs.bgu.ac.il/~ppl182/wiki.files/class/notebook/2.4SyntacticOperations.html#Rewriting-ASTs

שאלה 2: ייצוג אובייקטים ב L (20 נק)

א. מה הפלט של הקוד הבא: **(2 נקודות)**

```
(define make-equation
  (lambda (a b)
    (lambda (msg)
      (cond ((eq? msg 'left) a)
            ((eq? msg 'right) b)))))

(let ((p1 (make-equation 'var1 'var2))
      (p2 (make-equation 'var3 'var2)))
   (eq? (p1 'left) (p2 'right)))
```

#f _____

נתון הממשק הבא ב TypeScript לייצוג אובייקט של סטודנט:

```
interface Student {
    tag: "student";
    id:number;
    name:string;
    grades:number[]
};
const makeStudent = (i:number, n:string, gs:number[]): Student
=>
    ({tag: "student", id:i, name:n, grades:gs});
const isStudent = (s: any): s is Student => s.tag === "student";
const getId = (s: Student): number => s.id;
const getName = (s: Student): string => s.name;
const getGrades = (s: Student): number[] => s.grades;
```

במהלך הקורס ראינו שתי דרכים לממש 'אובייקט' בשפות ה L שפיתחנו: כרשימה, וכ-Closure.

ב. באיזו שפה מבין L1-L5 ניתן כבר לממש אובייקט כרשימה (=השפה האפשרית הפשוטה ביותר), ובאיזו ניתן לממש אובייקט כ-closure? נמק **(2 נקודות)**

כרשימה: We have lists in languages L3 and up

כ-closure: From languages L2 and up

Closures are sufficient without any other compound type and literal values to represent composite data values. Without closures, we need to introduce composite value constructors and accessors as primitives in the language to support compound data types.

ג. השלימו את החסר במימוש הבא של האובייקט Student בשפה 5L. יש לתייג, אם אפשר, את הטיפוסים השונים **(10 נקודות)**

The only type expressions we can use are the ones we have in the language L5 and up, and List, since we implemented the Pair type expression in homework, which may be expanded to list. The only type expressions we can use in (and are relevant to) this question are: boolean / number / string / symbol / List / T. Since the question indicates type annotations in the procedures – we must use at least L5 and up.
Note that there is no support for Type Unions in L5 – so that it is fine to use either Type Variables in places a union would be needed. Similarly for heterogeneous lists – we can write List(T) or just List – instead of List(T1 U T2).
We use the shorthand notation for list accessors (cadr, caddr…) to make it shorter instead of (car (cdr x)) etc.

Common mistakes:
1. Using the type expression "student" (it does not exist in our type language)
2. Using the type expression "any" (which does not exist in our type language). Must use “T” instead.
3. Writing the type expression "closure" -- there is no such type name in the type language, we have only procs: [P1 x P2 x ... x Pn -> RetType]. “Closure” is the name of value type – not of the expression types .
4. Writing a procedure type expression of the form [something x something x .. -> Union of Number, Boolean, etc]. We have no union type expressions in L5.
Type expressions which cannot be expressed in L5 could be left blank or using type variables (the question states “if possible”).

מימוש סטודנט כרשימה:

```
(define make-student-l
  (lambda ((i : number)
           (n : string)
           (gs : List(number))) : List
    (list 'student i n gs)))

(define student?-l
  (lambda ((s : T)) : boolean
    (eq? (car s) 'student)))

(define student->id-l
```

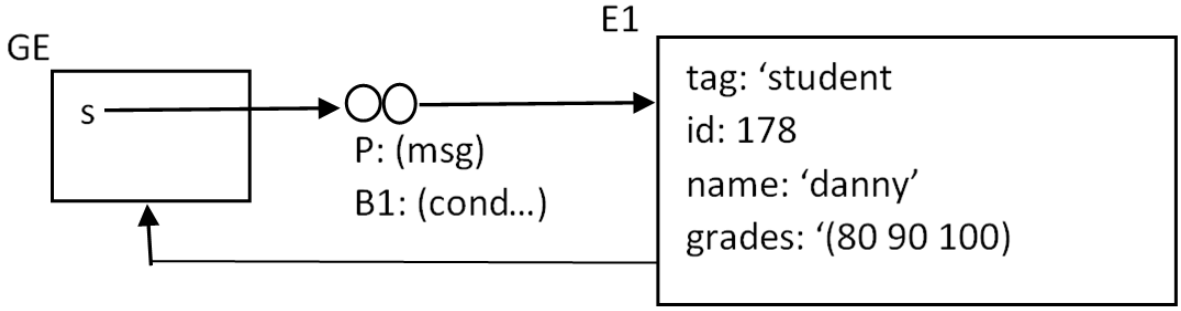
```
  (lambda ((s : List)) : number
    (cadr s)))

(define student->name-l
  (lambda ((s: List)) : string
    (caddr s)))

(define student->grades-l
  (lambda ((s: List)) : List
    (cadddr s)))
```

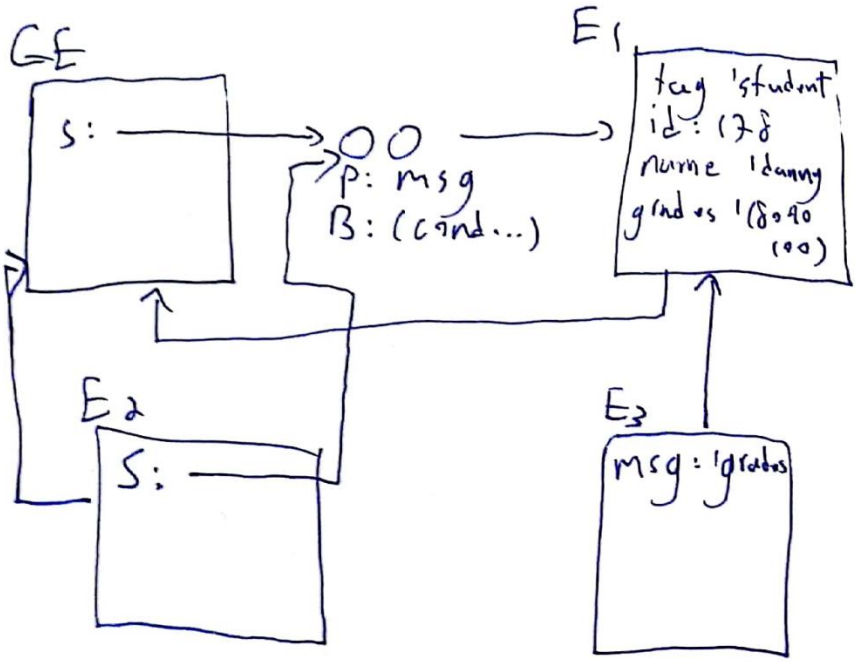
מימוש סטודנט כ-**closure**:

```
(define make-student-c
  (lambda ((i : number)
           (n : string)
           (gs : List))) : (symbol -> T)
    (lambda (msg)
      (cond ((eq? msg 'tag) 'student)
            ((eq? msg 'id) i)
            ((eq? msg 'name) n)
            ((eq? msg 'grades) gs)))))

(define student?-c
  (lambda ((s: (symbol -> T))) : boolean
    (eq? (s 'tag) 'student)))

(define student->id-c
  (lambda ((s : (symbol -> T))) : number
    (s 'id)))

(define student->name-c
  (lambda ((s : (symbol -> T)) : string
    (s 'name)))

(define student->grades-c
  (lambda ((s : (symbol -> T)) : List
    (s 'grades)))
```

ד. השלימו את דיאגרמת הסביבות עבור ההרצה של שני קטעי הקוד הבאים (האחד בגישת רשימה, והשני בגישת closure) באינטרפרטר של 5L הממומש ע"פ מודל הסביבות: **(6 נקודות)**

```
(define s (make-student-c 178 "danny" '(80 90 100)))
(student->grades-c s)
```

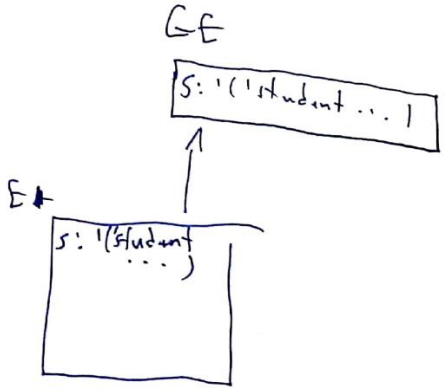





```
(define s (make-student-l 178 "danny" '(80 90 100)))
(student->grades-l s)
```

GE

s: ‘(‘student 178 ‘danny’ ‘(80 90 100))



שאלה 3: הסק טיפוסים (15 נקודות)

אלגוריתם הסק טיפוסים שנלמד בכיתה מייצר, בין היתר, משוואות טיפוסים לביטוי הנתון.
כזכור, המשוואות נגזרות מהסוג התחבירי של הביטויים: ביטויים פרימיטיביים (מספרים, ביטויים בוליאניים, אופרטורים פרימיטיביים), הגדרת פרוצדורה, הפעלת פרוצדורה/אופרטור, מבנה תנאי.
נתון הביטוי:

```
(
 (lambda (x y z) x y z)
 (f a)
 (* a b)
 c
)
```

ונתונים משתני הטיפוס לכל אחד מתתי הביטויים:

```
T1:   ((lambda (x y z) x y z) (f a) (* a b) c)
T2:   (lambda (x y z) x y z)
T3:   (f a)
T4:   (* a b)
T5:   f
T6:   a
T7:   b
T8:   c
T9:   x
T10:  y
T11:  z
T12:  *
```

השלימו את המשוואות הנוצרות מסוגי המבנים התחביריים הבאים (ב-ד), וציינו את הטיפוסים שניתן להסיק רק על פי המשוואות של מבנים אלו:

א. ביטויים פרימיטיביים

משוואות:

```
T12 = [number * number -> number]
```

טיפוסים מוסקים: לא ניתן להסיק דבר נוסף מעבר למה שמצויין במשוואות, כלומר רק את T12.

ב. ביטויים פרימיטיביים + הפעלת פרוצדורה/אופרטור **(5 נק)**

ביטויים פרימיטיביים – כמו בסעיף א

```
T12 = [number * number -> number]
```

הפעלת אופרטור - יש הפעלה אחת של אופרטור: `(* a b)`

```
T12 = T6 * T7 -> T4
```

הפעלת פרוצדורה – יש שתי הפעלות של פרוצדורות: `(f a)` , והביטוי כולו

```
T5 = T6 -> T3
T2 = T3 * T4 * T8 -> T1
```

טיפוסים מוסקים:

```
T6 = number
T7 = number
T4 = number
T5 = number -> T3
T2 = T3 * number * T8 -> T1
```

ג. הגדרת פרוצדורה + מבנה תנאי **(5 נק)**

משוואות:

הגדרת פרוצדורה – קיימת הגדרה של פרוצדורה אחת `(lambda (x y z) x y z)`

```
T2 = T9 * T10 * T11 -> T11
```

מבנה תנאי – אין מבנה תנאי בביטוי.

טיפוסים מוסקים: לא ניתן להסיק דבר נוסף מעבר למה שמצויין במשוואות, כלומר רק את T2.

ד. ביטויים פרימיטיביים + הגדרת פרוצדורה + הפעלת פרוצדורה/אופרטור **(5 נק)**

משוואות: כל המשוואות מסעיפים ב - ג

טיפוסים מוסקים:

```
T6 = T7 = T4 = T10 = number
T5 = number -> T3
T2 = T3 * number * T8 -> T1
T11 = T1 = T8
T3 = T9
```

שאלה 4 – CPS (26 נק)

א. מה ההבדל בין תהליך חישוב רקורסיבי לבין תהליך חישוב איטרטיבי? **(2 נק)**

בתהליך חישוב רקורסיבי ואיטרטיבי ישנה קריאה חוזרת ונשנית לאותה הפונקציה, כאשר בחישוב רקורסיבי תוצאת החישוב תלויה בקריאה הבאה, ועל כן יש לשמור את נתוני החישוב הנוכחי. השמירה נעשית באמצעות פתיחת מסגרות על מחסנית הקריאות, בסדר גודל של גודל הקלט.
בתהליך חישוב איטרטיבי אין תלות בחישוב בקריאה הבאה, ולכן מס' המסגרות במחסנית נותר קבוע.

ב. נתון הממשק הפונקציונלי הבא עבור עצים בינאריים: **(10 נק)**

```
empty?
node->value
node->left-child
node->right-child
```

נתון עץ בינארי אשר המידע אשר בצמתיו יכול להכיל פרוצדורות (מובטח כי הפרוצדורות אינן מקבלות פרמטר).
כתוב פונקציה בסגנון CPS אשר סורקת את העץ בינארי ב-DFS באמצעות הממשק הנ"ל ואשר מחזירה רשימה של התוצאות של ההפעלות של הפרוצדורות אשר בצמתים. צמת אשר אינו מכיל פרוצדורה, לא משפיע על הרשימה.
דוגמה:

```
(define my-tree
  (list 22
        (list (lambda () 15)
              '()
              (list 24
                    (list (lambda () 16)
                          '()
                          '())
                    '()))
        (list #t
              '()
              '())))

(tree-app$ my-tree (λ (x) x))
→ '(15 16)
```

השלימו את הפונקציה ואת הגדרת הטיפוס שלה:

```
; Purpose: determine whether all the nodes in t are procedures
; Signature tree-app$(t, cont)
; Type: [BinTree * [BinTree ->  T] -> T]
(define tree-app$
  (λ (t cont)
    (cond [(empty? t) (cont t)]
          [(procedure? (node->value t))
           (tree-app$
            (node->left-child t)
            (λ (tree-app-left)
              (tree-app$
               (node->right-child t)
               (λ (tree-app-right)
                 (cont
                  (append
                   tree-app-left
                   (cons ((node->value t)) tree-app-right
                    ))))))))]


          [else
           (tree-app$
            (node->left-child t)
            (λ (tree-app-left)
              (tree-app$
               (node->right-child t)
               (λ (tree-app-right)
                 (cont (append tree-app-left tree-app-right
                  ))))))]))))
```

ג. הציעו דרך לקרוא לפונקציית ה-CPS מהסעיף הקודם כך שיוחזר מספר הצמתים אשר מכילים פרוצדורות **(2 נק)**

```
(tree-apps$ my-tree length)
```

ד. מימשו פרוצדורה בסגנון success-fail שמפעילה את ה-success continuation במידה שכל הצמתים בעץ הם פרוצדורות, ואת ה-fail continuation במקרה שאחד מהצמתים אינו פרוצדורה. על הפרוצדורה לעבור על הצמתים בסדר depth-first ולעצור את המעבר מיד אחרי שהצמת הראשון שאינו פרוצדורה נמצא. **(12 נק)**

```
; Purpose: determine whether all the nodes in t are procedures
; Signature tree-sf$(t, success, fail)
; Type: [BinTree * [BinTree -> T1] * [Empty -> T2] -> T1|T2]
(define tree-sf$
  (λ (t success fail)
    (cond [(empty? t) (success t)]
          [(procedure? (node->value t))
          (tree-sf$
            (node->left-child t)
            (λ (left)
              (tree-sf$
               (node->right-child t)
               (λ (right)
                 (success t))
               fail))
            fail)]
          [else
           (fail)])))
```

שאלה 5 : תכנות לוגי (20 נק)

א. חשבו את ה-Unifiers הבאים: **(6 נק)**

```
unify[ t(X, f(a), X), t(g(U), U, W) ]

{X = g(f(a)), U = f(a), W = g(f(a))}


unify[ t(X, f(X), X), t(g(U), U, W) ]

Occur check fail when extending {X = g(U)} with {U = f(X)}
```

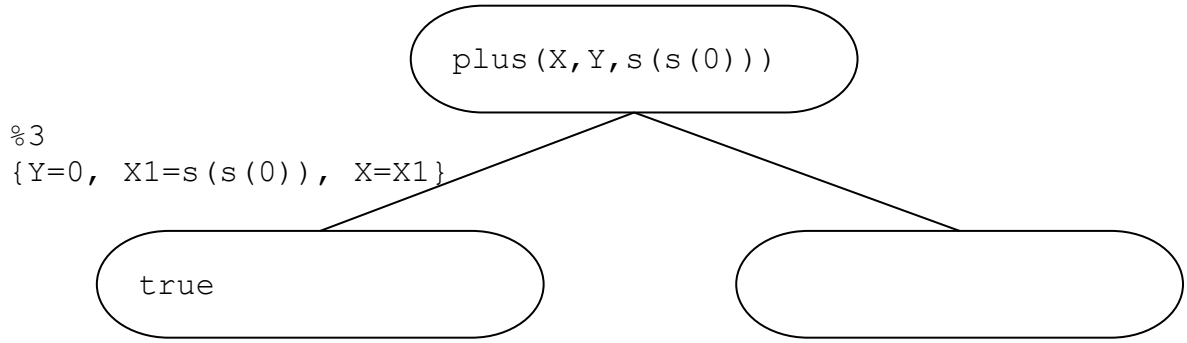
ב. היזכרו בהגדרות עבור מספרי צ'רץ': **(6 נק)**

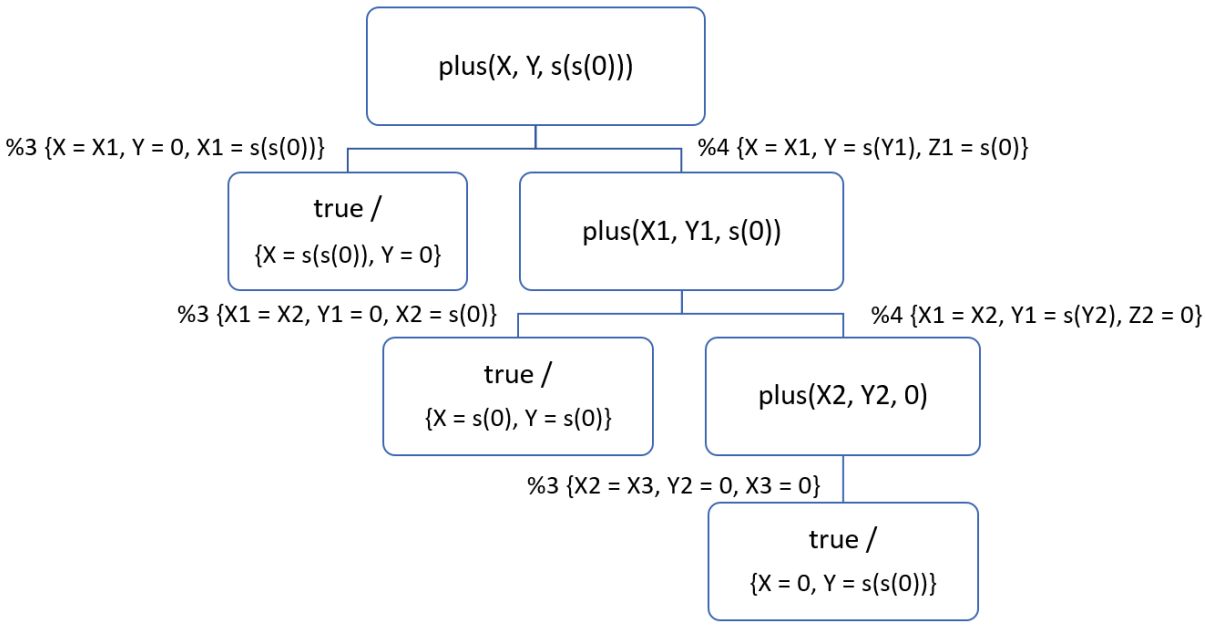
```
natural_number(0).                                      %1
natural_number(s(N)) :- natural_number(N).              %2

plus(X, 0, X).                                          %3
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).                   %4

times(_, 0, 0).                                         %5
times(X, s(Y), Z) :- times(X, Y, XY), plus(X, XY, Z).   %6
```

השלימו את עץ ההוכחה עבור השאילתא (התרשים בעמוד הבא)

```
plus(X, Y, s(s(0))).
```

The query can be read: find all pairs (X,Y) such that X+Y=2. We know the expected answers will be {X = 0, Y = s(s(0))}, {X = s(0), Y = s(0)} and {X = s(s(0)), Y=0}.


plus(X,Y,s(s(0)))
%3
{Y=0, X1=s(s(0)), X=X1}
true


plus(X, Y, s(s(0)))
%3 {X = X1, Y = 0, X1 = s(s(0))}
%4 {X = X1, Y = s(Y1), Z1 = s(0)}
true /
{X = s(s(0)), Y = 0}
plus(X1, Y1, s(0))
%3 {X1 = X2, Y1 = 0, X2 = s(0)}
%4 {X1 = X2, Y1 = s(Y2), Z2 = 0}
true /
{X = s(0), Y = s(0)}
plus(X2, Y2, 0)
%3 {X2 = X3, Y2 = 0, X3 = 0}
true /
{X = 0, Y = s(s(0))}

ג. האם עץ ההוכחה מהסעיף הקודם הוא עץ הצלחה או כישלון? נמקו **(2 נק)**

Success tree because there is at least one path leading to a success node.

ד. באמצעות מספרי צ'רץ' נגדיר מספרים רציונליים באופן הבא: **(6 נק)**

```
rational_number(q(X,Y)) :-
     natural_number(X),
     natural_number(Y),
     Y \= 0.
```

We know that rational numbers are of the form X/Y. We understand from this definition that X is the numerator and Y the denominator, because we check Y!=0.

הגדירו את פעולת החיבור עבור מספרים רציונליים:

```
We use the formula:
X1/Y1 + X2/Y2 = (X1*Y2 + X2*Y1)/ Y1*Y2
There is no need to simplify the rational number (compute gcd and
divide).

plus_r(q(X1,Y1), q(X2,Y2), q(X3,Y3)) :-

     times(X1, Y2, N1),
     times(X2, Y1, N2),
     plus(N1, N2, X3),
     times(Y1, Y2, Y3).
```