שפות תכנות – תרגיל 5

שאלה 1

a. We used this script:

```
<script>
        function map (f, inarray) {
        var out = [];
        for(var i = 0; i < inarray.length; i++) {
        out.push( f(inarray[i]) )
        }
        return out;
        }
        function reduce (f, inarray) {
        if (inarray.length <= 1) return;
        if (inarray.length == 2) return f(inarray[0], inarray[1]);
        var r = inarray[0];
        for(var li = 1 ; li < inarray.length ; li++) {
        r = f(r, inarray[li]);
        }
        return r;
        }
        function q1a()
        {
            var initialArray=[1,2,3,4,5];
            return reduce(function(x,y){return x+y;},map(function(x){return x*x;},initialArray));
        }
</script>
```

The relevant one line is:
reduce(function(x,y){return x+y;},map(function(x){return x*x;},initialArray))

Where the first action is map each element to it's square and the second action is reduce these map output elements to their sum.

b. We used this script:

```
<script>
        function map (f, inarray) {
        var out = [];
        for(var i = 0; i < inarray.length; i++) {
        out.push( f(inarray[i]) )
        }
        return out;
        }
        function reduce (f, inarray) {
        if (inarray.length <= 1) return;
        if (inarray.length == 2) return f(inarray[0], inarray[1]);
        var r = inarray[0];
        for(var li = 1 ; li < inarray.length ; li++) {
        r = f(r, inarray[li]);
        }
        return r;
        }
        function q1b()
        {
            var initialArray=[-1,-2,-3,0,5];
            return reduce(function(x,y){return x+y;},map(function(x){
                if(x>0)
                    return 1;
                else return 0;
                },initialArray));
        }
</script>
```

Map each positive number (x>0) to 1 and the other numbers to 0, and then summarize the array with reduce as we did in the first part of this question.

C. We used this script:

```
<script>
        function map (f, inarray) {
        var out = [];
        for(var i = 0; i < inarray.length; i++) {
        out.push( f(inarray[i]) )
        }
        return out;
        }
        function reduce (f, inarray) {
        if (inarray.length <= 1) return;
        if (inarray.length == 2) return f(inarray[0], inarray[1]);
        var r = inarray[0];
        for(var li = 1 ; li < inarray.length ; li++) {
        r = f(r, inarray[li]);
        }
        return r;
        }
        function q1c()
        {
            var initialArray=[[1,2],[3,4],[5,6],[7,8,9]];
            return reduce(function(x,y){return x.concat(y);},initialArray);
        }
</script>
```

We have to use only reduce function between the elements: the reduce function is concat, which returns a flatten array of two arrays.

Map function is not needed here because the elements are already arrays as required.

שאלה 2

```
var x = 5;
function f(y) { return (x + y) - 2 };
function g(h) { var x = 7; return h(x) };
{ var x = 10; z = g(f) };
```

סעיף a

הערך הוא 15

סעיף b

x=10, y=7

סעיף c

y הוא הערך שהועבר לפונקציה f כארגיומנט. הפונקציה נקראה מתוך הפונקציה g שהעבירה ל-f (באופן כללי ל-h, אבל בריצה שלנו הפונקציה f הועברה ל-g כארגיומנט) את x=7 שהוגדר לוקלית אצלה.

סעיף d

x הוא הערך 10 ש"הוגדר" בתוך הסוגריים המסולסלים האחרונים. הפונקציה f משתמשת **בערך העדכני ביותר (בעת ההרצה שלה)** של x, ובהגדרה של x **ה"קרובה ביותר" שהוגדרה בscope של ההגדרה f** (או בסקופים מעליה). מכיוון ש {} כלשעצמו ב-JS לא פותח scope חדש, ה-x בשורה הראשונה והאחרונה שניהם מוגדרים באותו scope, ובאותו אחד כמו של f, כלומר שניהם אותו ה-x (הפעולה השניה על x היא בעצם רק השמה).
לכן x מחושב עם הערך העדכני ביותר, לפני הקריאה לפונקציה g, מבין השניים (5,10) - שהוא 10.
(נשים לב ש<u>var x=7</u> שמוגדר לוקלית בתוך scope של פונקציה g, אמנם מוגדר לפני הקריאה לפונקציה f, אבל <u>**ההגדרה שלו** היא לא בscope של ההגדרה של f ולא מעליה</u> (אלא בסקופ פנימי) ולכן אין שימוש בו).

```
(function () {
	var x = 5;
	(function () {
		function f(y) {return (x + y) - 2};
		(function () {
			function g(h) {var x = 7; return h(x)};
			(function () {
				var x = 10; z = g(f);
			})();
		})();
	})();
})();
```

סעיף e

הערך הוא 10.

סעיף f

x=5, y =7

סעיף g

(כמו בדוגמא הקודמת) y הוא הערך שהועבר לפונקציה f כארגיומנט. הפונקציה נקראה מתוך הפונקציה g שהעבירה ל-f (באופן כללי ל-h, אבל בריצה שלנו הפונקציה f הועברה ל-g כארגיומנט) את x=7 שהוגדר לוקלית אצלה.

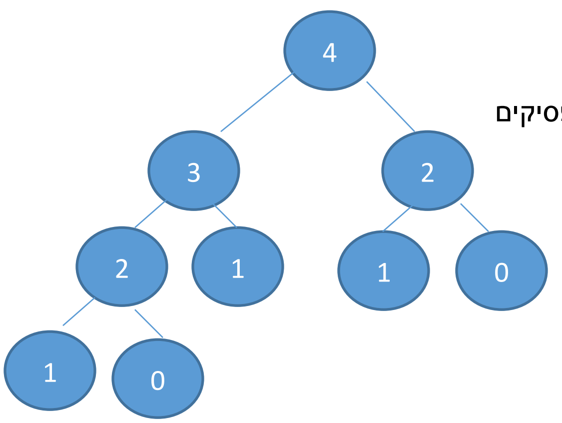
סעיף h

בניגוד לדוגמא הקודמת, כעת שורת ההרצה `z = g(f)`, מוגדרת בתוך scope של פונקציה, ולכן כאשר נקרא לפונקציה f (דרך הפונקציה g) היא לא תכיר את המשתנים בscopes הפנימיים שקראו לה. ההגדרה הקרובה ביותר אליה כעת (שנמצאת מעליה) היא var x = 5.

שאלה 4

סעיף a

ב- fibonacci הסיבוכיות זמן ריצה היא O(n) ובnaive_fibonacci היא O(2^n).
הסיבה היא ש Fibonacci משתמש ב- memoization – כלומר הוא שומר חישובים של מספרים שהוא כבר חשב.
נקח דוגמא קטנה, כדי לחשב את מספר פיבונציי 4 נוצר העץ הבא:



על מנת לחשב את מספר פיבונצי נצטרך לדעת את שני המספרים הקודמים לו ולכן נוצר העץ הזה כאשר 0,1 ידועים לנו מראש, ולכן כאשר מגיעים עליהם מפסיקים את הירידה בעץ.

במקרה שלנו בשני הפונקציות הרקרוסיה הולכת קודם לעץ השמאלי (לn-1). כלומר כאשר חישבנו כבר את תת העץ השמאלי, כחלק מהחישוב גם חישבנו את מספרי פיבונאצי 2,3.

ההבדל בין השיטה בלי memo לעם memo, היא מה קורה כאשר מגיעים למספר שלא הוגדר לנו מראש אבל כבר חישבנו אותו במהלך הריצה על העץ.
במקרה של naive_fibonacci כאשר נעבור לחישוב תת העץ הימני נצטרך לחשב עכשיו את 2 מחדש.
לעומת זאת ב- Fibonacci נשמור את החישוב של 2, ונוכל מיד לחזור.
(בדוגמאות עם מספרים יותר גדולים הדבר כמובן קורה גם בתוך כל תת עץ של השורש).

סעיף b

המטרה של המשתנה memo היא בדיוק לשמור את מספרי פיבונאצי שחישבנו כבר על מנת לא לחשב אותם שוב כאשר נתקל בהם בריצה על העץ

סעיף c

המשתנה memo מוגדר בתוך הscope של הפונקציה ההאונונימית. אך מיכוון שהפונקציה האנונונימית מחזירה פונקציה (שמוגדר בתוכה) ומשתמש בmemo, נוצר העתק על הheap של memo. זאת מיכוון שבסיום הריצה של הפונקציה האנונמית המשתנה memo שלה שהוגדר על המחסנית נהרס, ולכן הפונקציה המוחזרת לא יכולה לגשת אליו.

סעיף d

לאורך כל הריצה של התוכנית (או עד להחלטתו הלא צפויה של ה-GC), זאת מיכוון שאנחנו לא יודעים מתי הפונקציה תהיה בשימוש, לכן נצטרך להחזיק את memo לזמן לא ידוע מראש.

סעיף e

1) הפונקציה האנונמית מאפשרת לנו לא לחשוף את memo לכלל התוכנית (הוא נגיש רק מתוך הפונקציה)
2) מיכוון שmemo לא נמחק כאשר אנחנו מסיימים את ריצת הפונקציה, ונוצר מחדש כאשר נכנסים אליה, נשמרים לנו כל מספרי פיבונאצי שחושבו כבר על ידי ריצות קודמות של הפונקציה.
לדוגמא אם הרצנו פעם ראשונה Fibonacci(10) – וחושבו כל מספרי פיבונאצי מ-2 עד 10. כעת בהרצה השניה של Fibonacci(10) לא יחושב דבר, וישר יחזור מספר פיבונאצי העשירי!

שאלה 5

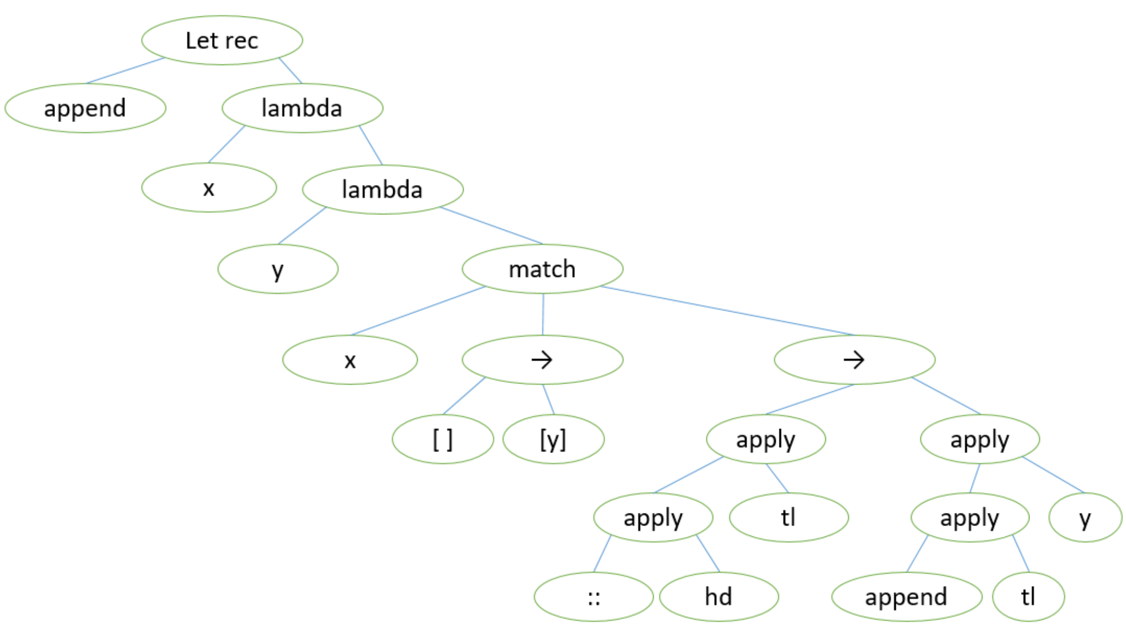
סעיף a

נבצע את תהליך הסקת הנתונים על הפונקציה הנתונה.

1. בניית parse tree

מבנה חולית let rec – כמו שראינו בתרגול, הבן השמאלי הוא שם הפונקציה והבן הימני הוא ביטוי הלמדה שמגדיר את הפונקציה.

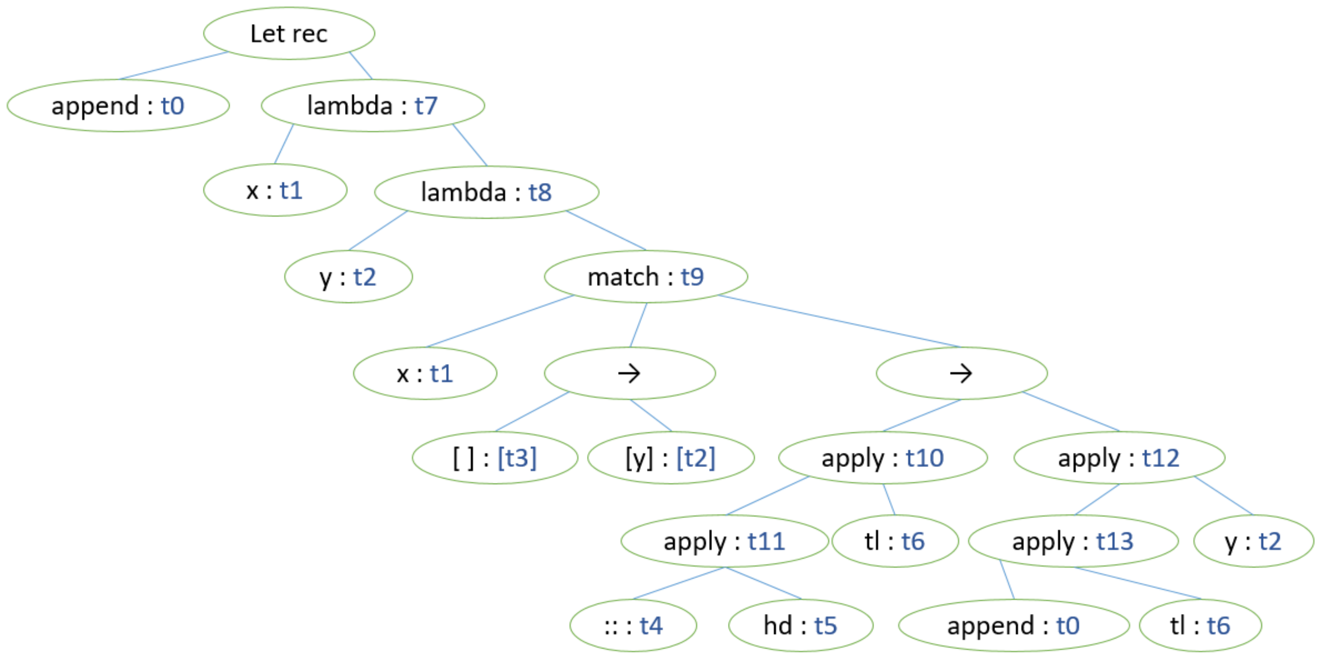
מבנה חולית match – לא ראינו בכיתה ולכן החלטנו להגדיר זאת באופן הבא: הבן השמאלי הוא המשתנה עליו מבצעים את הpattern matching, שאר הבנים הם אפשרויות להתאמה. כל אפשרות כזאת מיוצגת ע"י חולית "->", שהבן השמאלי שלה הוא התאמה לpattern והבן הימני הוא הערך שה-match יחזיר באפשרות זו.



2. <u>הוספת טיפוסים לעץ</u>

נוסיף טיפוסים לחוליות.

(לחוליות ה-"->" לא נוסיף טיפוס. בשלב הבא נדאג לקשר בין הטיפוסים של החוליות השונות בתת העץ match, על פי הסמנטיקה של match).



3. <u>יצירת אילוצים</u>

t0 = t7 // let rec

t7 = t1 -> t8 // lambda expression

t8 = t2 -> t9 // lambda expression

t9 = [t2]

t9 = t12

match result

t1 = [t3]

t1 = t10

match pattern

t11 = t6 -> t10 // application

t4 = t5 -> t11 // application

t4 = a -> a -> [a] // :: is a built-in operator

t13 = t2 -> t12 // application

t0 = t6 -> t13 // application

4. <u>פתרון האילוצים באמצעות unification</u>

נפתור את המשוואות ונקבל:

t0 = [a] -> t2 -> [t2]

t1 = [a]

t3 = a

t4 = a -> [a] -> [a]

t5 = a

t6 = [a]

t7 = [a] -> t2 -> [t2]

t8 = t2 -> [t2]

t9 = [t2]

t10 = [a]

t11 = [a] -> [a]

t12 = [t2]

t13 = t2 -> [t2]

5. <u>מציאת הטיפוס של הפונקציה הנתונה</u>

הטיפוס של append הוגדר להיות t0, ולכן:

append : [a] -> t2 -> [t2]

או אם נסמן t2 = b בשביל האחידות:

append: [a] -> b -> [b]

כאשר a, b הם טיפוסים כלליים.

סעיף b

כן, יש משהו שמצביע על טעות. אמנם הצלחנו להסיק את הטיפוס של append, אבל קיבלנו שהיא פונקציה שמקבלת רשימה עם איברים מטיפוס a ואיבר נוסף מטיפוס b. append אמורה לשרשר איבר לרשימה, ולכן היינו מצפים לקבל append: [a] -> a -> [a]. כמו כן, גם בלי לדעת מה מטרת הפונקציה append, זה נראה משונה שהטיפוס שהיא מחזירה כלל לא מתייחס לטיפוס של הארגומנט.

סעיף c

תיקון הפונקציה:

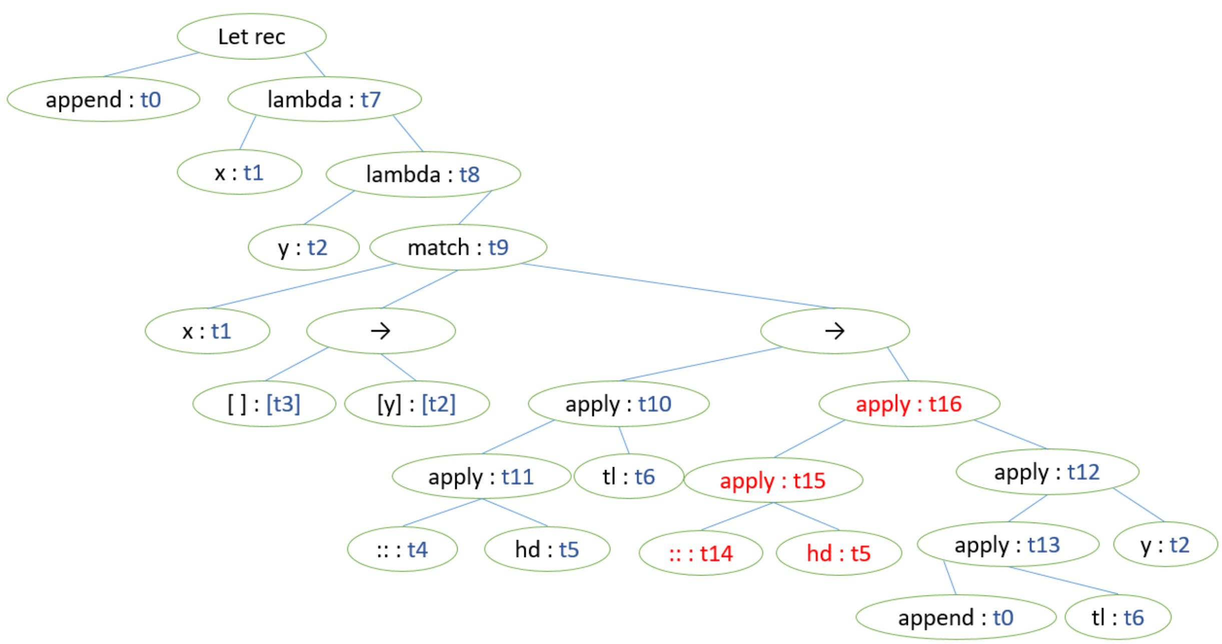
```
let rec append x y =
        match x with
        | [ ]        -> [y]
        | hd :: tl  -> hd :: (append tl y)
```

סעיף d

לאחר התיקון נקבל:

append: [a] -> a -> [a]

ההבדל בהרצה של אלגוריתם הסקת הטיפוסים הוא שלאחר התיקון נקבל עץ גדול יותר:



נקבל את האילוץ t15 = t12 -> t16 (application), וכן t15 = [a] -> [a] (בגלל ה-:: והטיפוס של hd שנשאר כמו מקודם). בסך הכל נסיק ש-t12 = [a].

תת העץ של t12 נשאר כפי שהוא ולכן נקבל עדיין t12 = [t2]. מכאן ש-t2 = a.

לכן t0 = [a] -> t2 -> [t2] מסעיף a יהפוך ל-t0 = [a] -> a -> [a], כפי שרצינו.