גיליון תשובות מספר נבחן:_____

על תשובות ריקות יינתן 20% מהניקוד!

שאלה 1 [35 נקודות]

1.1. [4 נקודות]

```
<cexp> ::= ...
       | (for <var-decl> <cexp> <cexp> <cexp>) // for-exp(var:var-decl,
low:cexp, high:cexp, body:cexp)
```

1.2. [6 נקודות]

```
export interface ForExp {tag: "ForExp"; var: VarDecl; low: CExp; high: CExp;
body: CExp };
export const makeForExp = (var_: VarDecl, low: CExp, high: CExp, body: CExp):
ForExp => ({tag: "ForExp", var: var_, low: low, high: high, body: body});
export const isForExp = (x: any): x is ForExp => x.tag === "ForExp";
```

1.3. [10 נקודות]

```
const evalFor = (exp: ForExp, env: Env): Value | Error => {
    const low = applicativeEval(exp.low, env);
    const high = applicativeEval(exp.high, env);
    for (let i = low; i < high; i++) {
        applicativeEval(exp.body, makeExtEnv([exp.var.var], [i], env));
    return applicativeEval(exp.body, makeExtEnv([exp.var.var], [high], env));
}
```

1.4. [10 נקודות]

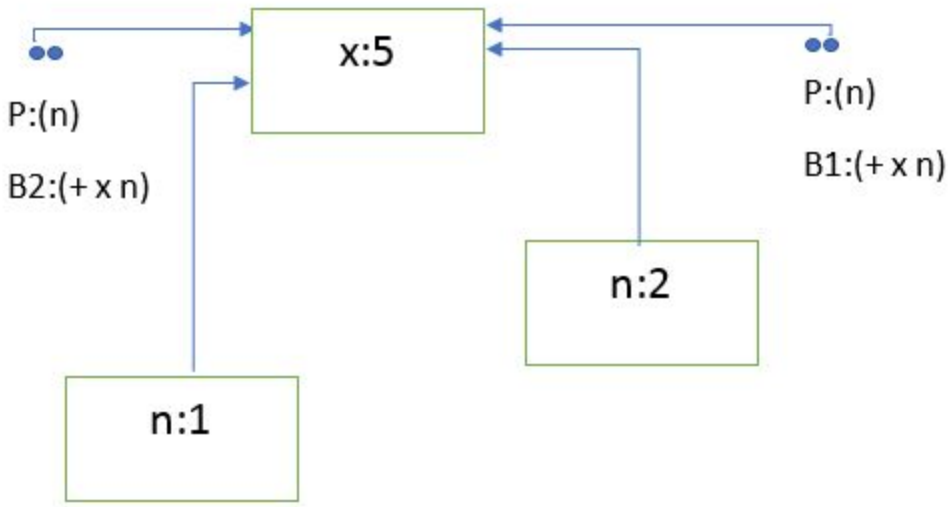
1.4.1 [2 נקודות]

```
((lambda (n) (* n n)) 1)
((lambda (n) (* n n)) 2)
```

1.4.2 [8 נקודות]

```
const For2Apps = (exp: ForExp, env: Env): AppExp[] => {
    const ret: AppExp[] = [];
    const low = applicativeEval(exp.low, env);
    const high = applicativeEval(exp.high, env);
    for (let i = low; i <= high; i++) {
        ret.push(makeAppExp(makeProcExp([exp.var], [exp.body]),
                                [makeNumExp(i)]));
    }
    return ret;
}
```

1.5. [5 נקודות] env diagram



הערה: התקבלו גם איורים ללא הclosures, אך מי שצייר רק closure אחד - ירדה לו נקודה.

שאלה 2 [25 נקודות]

2.1 [6 נקודות]

2.1.1

```
interface I3 = {a:number, b: string, c: boolean}
```

2.1.2

```
type R3 = range(2,10)
```

2.1.3

```
type F3 = ( a: number | string ) => boolean
```

Note: When we compute the intersection of two functional types, the argument receives a type which is the union of the arguments of the intersected types, while the return type is the intersection.

2.2 [8 נקודות]

2.2.1

כתבו רק השורות החדשות או את אלה שצריך לשנות כדי לתמוך ב-range-te ו-inter-te:

Changes in the BNF:

```
<composite-te> ::= <proc-te> | <tuple-te> | <inter-te> | <range-te>
<non-tuple-te> ::= <atomic-te> | <proc-te> | <tvar> | <inter-te> | <range-te>
```

Changes in the type definitions:

```
interface InterTExp { tag: "InterTExp"; components: NonTupleTExp[]; }
const makeInterTExp = (components: NonTupleTExp[]): InterTExp =>
      ({tag: "InterTExp", components: components});
const isInterTExp = (x: any): x is InterTExp => x.tag === "InterTExp";

interface RangeTExp {tag: "RangeTExp"; low: LitExp; high: LitTexp; };
const makeRangeTExp = (low: LitExp, high: LitExp): RangeTExp =>
    ({tag: "RangeTExp", low: low, high: high});
const isRangeTExp = (x: any): x is RangeTExp => x.tag === "RangeTExp";

type compoundTExp = …. | InterTExp | RangeTExp;
const isCompoundTExp = (x: any): x is CompoundTExp =>
    … | isInterTExp(x) | isRangeTExp(x);

type nonTupleTExp = … | InterTExp | RangeTExp;
const isNonTupleTExp = (x: any): x is NonTupleTExp =>
    … | isInterTExp(x) | isRangeTExp(x);
```

2.2.2

(number & boolean) →

```
{tag: "InterTExp", components: [{tag: "BooleanTExp"}, {tag: "NumberTExp"}]}
```

[The components list must be sorted]

(number & (string & boolean)) →

```
{tag: "InterTExp",
 components: [{tag: "BooleanTExp"}, {tag: "NumberTExp"}, {tag: "StringTExp"}]}
```

w

[The components list is flattened]

(number & (boolean & number)) →

```
{tag: "InterTExp", components: [{tag: "BooleanTExp"}, {tag: "NumberTExp"}]}
```

[The components list has no duplicates]

```
((range 1 10) & number) →

{tag: "InterTExp",
 components: [{tag: "NumberTExp"},
              {tag: "RangeTExp", low: {tag: "LitExp", value: 1},
                                 high: {tag: "LitExp", value: 10}]}
```

NOTE: The transformation of the components into a list that is flat / sorted and with no duplicates is a **syntactic** transformation that occurs at **parsing time**, when constructing the AST of the TExp. We do **not** perform any **semantic transformation** at this stage. For example, the fact that the intersection (number & boolean) is empty is a **semantic inference** -- it depends on the specific rules of type checking that are defined in the type system. Such answers were not accepted for this question (they got partial credit when used consistently).

2.3 [5 נקודות]

1. AtomicType1 and AtomicType2 are compatible only if they are identical

2. AtomicType1 and (T1 & T2) are compatible if:

   AtomicType1 is compatible with T1 and AtomicType1 is compatible with T2
   Alternatively:
   AtomicType1 = T1 and AtomicType1 = T2

   NOTE: AtomicTypes are boolean, number, string. Atomic Types are only compatible with themselves (for example, number is not included in any range or any procedure or any intersection which is not equal to number). Hence, the constraint that leads to the correct answer.

3. AtomicType1 and ProcExpT2 are not compatible.

4. (T1 & T2) and AtomicType2 are compatible if:

   T1 is compatible with AtomicType2 or T2 is compatible with AtomicType2

   NOTE: The important example to keep in mind is:
   `range(1,10) & range(2,12) is compatible with number`

This type checking rule breaks up the intersection on the left hand side with two constraints on each of the components. The new constraints must then be checked recursively.
In general, when T1 is included in AtomicType2 and T2 is included in AtomicType2, then the intersection T1 & T2 must be included in AtomicType2.
If only one of them is included in AtomicType2 - then the intersection is also included in AtomicType2 - including the case where the intersection is empty.
The complicated case is when none of T1 and T2 is included in AtomicType2. In this case, logically their intersection could still be included in AtomicType2. In this case, we must perform a case by case analysis. Only range-types can be strictly included in an Atomic Type (a procedure cannot be included in an atomic type). Hence this last case is not possible.

5. (T11 & T12) and (T21 & T22) are compatible if:

   (T11 & T12) is compatible with T21 and (T11 & T12) is compatible with T22

   NOTE: For the left-hand side to be included in the intersection, it must be included in each of T21 and T22 separately. This is always true for any sets.
   It is also sufficient to break up the complex case into two simpler case and to continue type checking recursively the two resulting type constraints.

6. (T1 & T2) and ProcTExp2 are compatible if: <no need to specify>

7. ProcTExp1 and AtomicType2 are not compatible.

8. ProcTExp1 and (T1 & T2) are compatible if: <no need to specify>

9. ProcTExp1 and ProcTExp2 are compatible (unchanged - no need to specify).

NOTE:
The goal of type checking rules is to provide an effective computation method to check that two type expressions are compatible. It must allow structural induction on the AST of the Type Expressions involved in the rule.
Answers which were paraphrases of the left hand side do not allow this structural induction. For example:
AtomicType1 and (T1 & T2) are compatible

If AtomicType1 is a subtype of (T1 & T2)
This is not informative.
Or:
For all x in AtomicType1, x is in T1 and x is in T2
This is not a statement that can be tested effectively (because there can be an infinite number of values in AtomicType1).

When grading, we verified that the terminology is compatible with the fact that type expressions denote set of values. Statements such as “T1 is in T2” are not acceptable.

2.4 [6 נקודות]

```
(L5 (define f (lambda ((x : (range 1 10)) : boolean #t)))
    (define g (lambda ((y : (range 0 5)) : number 0))
    (define (p : ((range 1 10) & (range 0 5))) 2)
    (and (f p)
         (> (g p) 0)))
```

NOTE: The key point here is that the program MUST force the type inference to use an intersection. We obtain this constraint because we pass the same variable to two functions with different types for their argument -- this creates the type inference that p must have a type within an intersection.

שאלה 3 [30 נקודות]

3.1 [4 נקודות] מה ההבדל בין תהליך חישוב רקורסיבי לתהליך חישוב איטרטיבי?

בתהליך חישוב רקורסיבי ואיטרטיבי ישנה קריאה חוזרת ונשנית לאותה הפונקציה, כאשר בחישוב רקורסיבי תוצאת החישוב תלויה בקריאה הבאה, ואילו בתהליך חישוב איטרטיבי אין תלות בחישוב בקריאה הבאה.

התלות בקריאה הבאה מתבטאת בפתיחה של מסגרות במחסנית הקריאות, כלומר בחישוב רקורסיבי נפתחות מסגרות במחסנית ואילו בחישוב איטרטיבי לא.

טעויות נפוצות:

- בחישוב רקורסיבי נפתחות **סביבות של מודל הסביבות**, ואילו בחישוב איטרטיבי לא.
- מציינים כי בחישוב רקורסיבי נפתחות מסגרות במחסנית הקריאות, אבל לא מציינים את הסיבה לכך.

3.2 [4 נקודות] תנו תרחיש שימוש (use case) בו עדיף שימוש ברשימה עצלה על פני רשימה רגילה, והסבירו מדוע הוא עדיף (להסביר במילים, אין צורך בקוד)

כאשר מעבדים רשימות אינסופיות (או גדולות מאוד), אין אפשרות לשמור את הרשימות בזיכרון, ולכן שימוש ברשימות עצלות הוא הכרחי.

3.3 [10 נקודות]

התקבלו שתי גרסאות: גרסה בה האופרטור אינו ב-CPS (זו הייתה כוונת השאלה) ואופרטור ב-CPS.

גרסה בה האופרטור **אינו** ב-CPS:

```
;; Type: [[T1 -> Boolean] * LZL(T1) * [List(T1) -> T2] -> T2]
(define take-while$
  (lambda (p lzl cont)
    (if (or (empty-lzl? lzl) (not (p (head lzl))))
        (cont empty)
        (take-while$ p
                     (tail lzl)
                     (lambda (tail-res)
                       (cont (cons (head lzl) tail-res)))))))
```

גרסה בה האופרטור **כן** ב-CPS:

```
(define take-while$$
 (lambda (pred$ lzl cont)
   (if (empty-lzl? lzl)
       (cont empty)
       (pred$
        (head lzl)
        (lambda (pred-head)
          (if (not pred-head)
              (cont empty)
              (take-while$$
               pred$
               (tail lzl)
               (lambda (take-tail)
                 (cont (cons (head lzl) take-tail))))))))))
```

טעויות נפוצות:

- טעויות בחתימה
- בגרסת ה-"לא CPS": בדיקת הפרדיקט בתוך ה-continuation. זה יגרום ללולאה אינסופית.
- בתנאי הבודק את האיבר באמצעות הפרדיקט, במקום להחזיר רשימה ריקה קוראים לפונקציה שוב רק בלי לשרשר את האיבר. זה יוצר פונקציית filter במקום הפונקציה הנדרשת

3.4 [10 נקודות]

```
function takeWhile<T>(gen: Iterable<T>,
                      pred: (x: T) => boolean): T[] {
   const result: T[] = [];
   for (let x of gen) {
       if (pred(x)) {
           result.push(x);
       } else {
           break;
       }
   }
   return result;
}
```

גרסאות נוספות שהתקבלו:

- שימוש ב-while במקום ב-for
- שימוש ברקורסיה במקום בלולאה

טעויות נפוצות:

- מימוש הפונקציה כ-filter
- אי-עצירת האיטרציה כאשר איבר לא מקיים את הפרדיקט (מה שמוביל ללולאה אינסופית)

3.5 [2 נקודות] מה יקרה אם הקלט לפונקציות בסעיפים 3.3 ו-3.4 יהיה רשימות עצלות/generators אינסופיים כאשר כל האיברים מקיימים את הפרדיקט?

החישוב לא יסתיים כיוון שתתרחש לולאה אינסופית.

התקבלו גם תשובות שציינו שהתוכנית יכולה לקרוס עקב מחסור בזיכרון הנובע מבניית הרשימה.

שאלה 4 [20 נקודות]

סעיף א [5 נקודות]

```
% Signature: natural(X)/1
% Purpose: X is a (list representation) of a number.
natural([]).
natural([X|Xs]) :- natural(Xs).
```

טעויות נפוצות: שימוש לא נכון ב Lists.

סעיף ב [5 נקודות]

```
% Signature: smaller(X, Y)/2
% Purpose: X<Y.

smaller([], [X|Xs]).
smaller([X|Xs], [Y|Ys]) :- smaller(Xs, Ys).
```

טעויות נפוצות:

- שימוש בסימבול [X] בכלל הראשון שהופך את הפתרון **ללא נכון** שכן [X] מייצג רשימה לא ריקה באורך 1.
- העתקת הפתרון של מספרי צ'רצ' מהכיתה ללא התייחסות לכך שהדרישה היא לייצוג אחר של מספרים טבעיים.
- פתרון שהיה נכון מלבד עבור שני קבוצות ריקות - שימו לב בפתרון שלנו השורה הראשונה מבטיחה שקבוצה ריקה קטנה מכל קבוצה לא ריקה.
- התקבלו גם וריאנטים של הפתרון שדרשו שהמשתנים הם natural בהנחה ועשו זאת נכון.

סעיף ג [5 נקודות]

```
% Signature: plus(X, Y, Z)/3
% Purpose: Z is the sum of X and Y.

plus([], [], []).
plus([], [_|Ys], [_|Zs]) :- plus([], Ys, Zs).
plus([_|Xs], Ys, [_|Zs]) :- plus(Xs, Ys, Zs).
```

NOTE: This is almost equivalent to the append predicate - but the difference is that the elements don't need to be the same - only the lengths of the lists are constrained.
For example:
?- plus([1,2], [1,2], [4,4,4,4]).
should return true.

A solution identical to append was accepted:

```
plus([], Y, Y).
plus([X|Xs], Y, [X|Zs]) :- plus(Xs, Y, Zs).
```

סעיף ד [5 נקודות]

```
% Signature: times(X,Y,Z)/3
% Purpose: Z = X*Y

% 0 x Y = 0
times([], _, []).
% (Xs+1) * Y = Z if (Xs * Y) + Y = Z
times([_|Xs], Y, Z) :- times(Xs, Y, XY), plus(XY, Y, Z).
```