גיליון תשובות מספר נבחן:_____

על תשובות ריקות יינתן 20% מהניקוד!

שאלה 1 [25 נקודות]

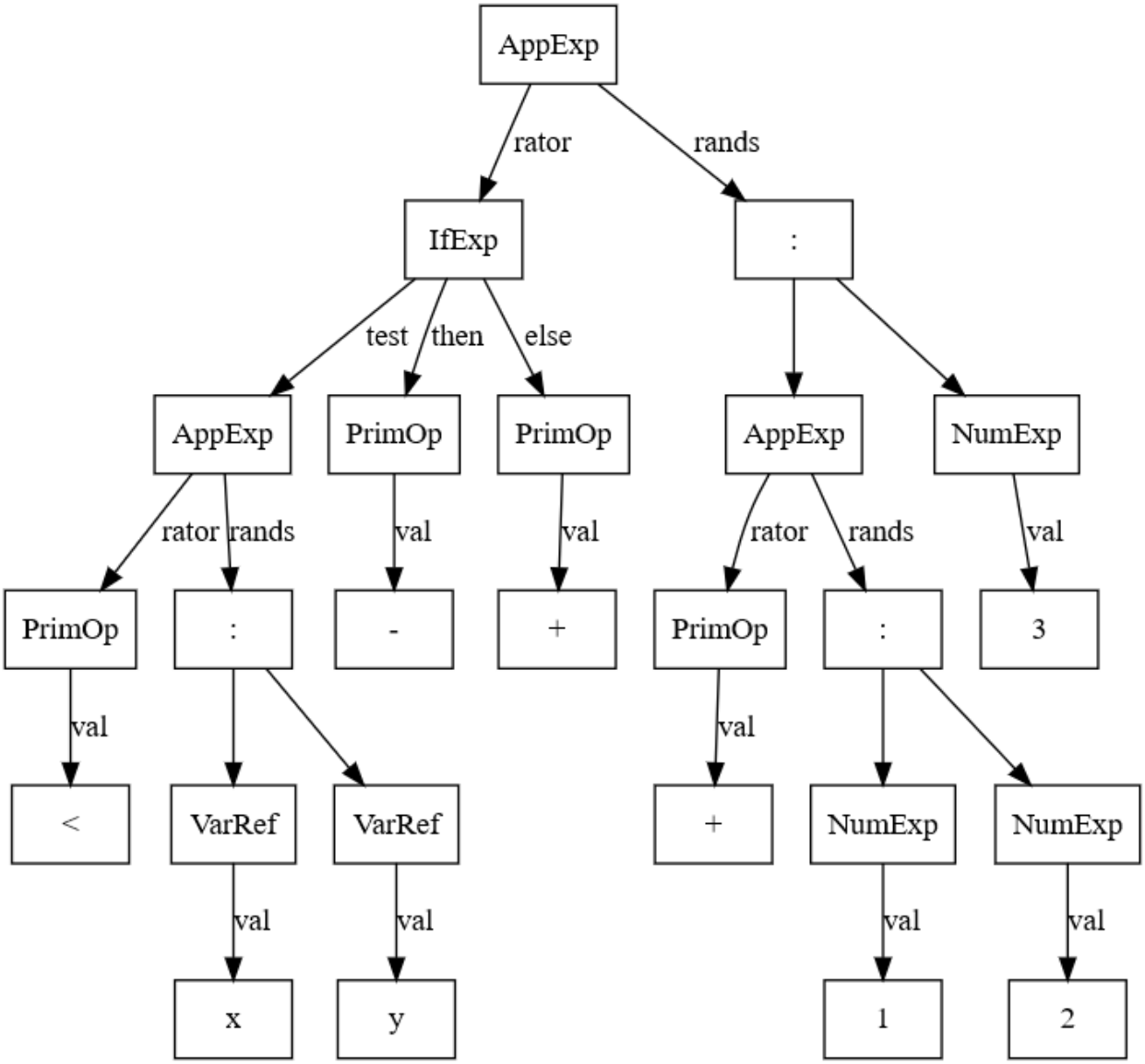
א. [9 נקודות]

```
const curry: <T1,T2,T3>(f: (x: T1, y: T2) => T3) => (x: T1) => (y: T2) => T3
= f => x => y => f(x, y);
```

ב. [9 נקודות]

```
const isShape = (x: any): x is Shape => isCircle(x) || isRectangle(x);
const isCircle = (x: any): x is Circle => x.tag === ‘circle’;
const isRectangle = (x: any): x is Rectangle => x.tag === ‘rectangle’;
```

ג. [7 נקודות]


AppExp
rator
rands
IfExp
:
test
then
else
AppExp
PrimOp
PrimOp
AppExp
NumExp
rator
rands
val
val
rator
rands
val
PrimOp
:
-
+
PrimOp
:
3
val
val
<
VarRef
VarRef
+
NumExp
NumExp
val
val
val
val
x
y
1
2

שאלה 2 - תכנות פונקציונלי [30 נקודות]

א. [10 נקודות]

```
; Signature: reverse(lst)
; Type: [List -> List]
; Purpose: return the items of a given list in an opposite order
; Pre-conditions: true
; Tests: (reverse ‘(1 2 3)) → ‘(3 2 1), (reverse ‘()) → ‘()
(define reverse
  (lambda (lst)
    (if (eq? lst '())
        '()
        (append (reverse (cdr lst)) (list (car lst))))))
```

ב. [10 נקודות]

```
; Signature: map2(f,lst1,lst2)
; Type: [[T1 * T2 -> T3] * List(T1) * List(T2) -> List(T3)]
; Purpose: apply a given binary procedure on item pairs of two given lists
; Pre-conditions: (length lst1) == (length lst2)
; Tests: (map2 + ‘(1 2 3) ‘(4 5 6)) → ‘(5 7 9)
(define map2
  (lambda (f lst1 lst2)
    (if (eq? lst1 '())
        '()
        (cons (f (car lst1) (car lst2))
              (map2 f (cdr lst1) (cdr lst2))))))
```

ג. [10 נקודות]

```
; Signature: derive(f,dx)
; Type: [[Number -> Number] * Number -> [Number -> Number]]
; Purpose: return a function which calculates an approximated derivation
;          of a given function.
; Pre-conditions: dx != 0
; Tests: ((derive square 0.001) 2) → 4.000999999999699
(define derive
  (lambda (f dx)
    (lambda (x)
      (/ (- (f (+ x dx)) (f x)) dx))))
```

שאלה 3, תרגום קוד [30 נקודות]

א.1 [6 נקודות]

- דוגמא לשתי שפות כאלה

sourceL: L2
targetL: L2 + let form

- תנאי המאפשר תרגום: טרנספורמציה תחבירית של המבנה החסר למבנה הקיים בשפת המטרה.
- דוגמא: ניתן להמיר את let לאפליקציה של lambda.

א.2 [6 נקודות]

- דוגמא לשתי שפות כאלה

sourceL: L2
targetL: L2 + primitive operator '^' (the power operation)

- תנאי המאפשר תרגום: ניתן לממש את האופרטור הפרימיטיבי כפרוצדורת משתמש בשפת המטרה.
- דוגמא: ניתן לממש את פעולת החזקה בעזרת פעולת כפל.

ב.1 [5 נקודות]

אבחנות:
- הפרוצדורה l2ToPython מתרגמת רק את הביטוי האחרון של גוף הפרוצדורה, תוך השמטת כל השאר.
- בסמנטיקה של L2, הערך של הפעלת פרצודורה הוא ערכו של הביטוי האחרון ב body.
- ב L2 אין side effects (כמו פעולות השמה, הדפסה, וכדומה).
- ב L2 עשויים להופיע ביטויים הגוררים שגיאת זמן ריצה (כמו חלוקה ב-0) או לולאה אינסופית (כמו הפרוצדורה הרקורסיבית loop שהודגמה בכיתה).

האבחנה הראשונה עשויה לפגוע בשקילות הסמנטית, כי לא מתרגמים את כל הביטויים ב body, אך האבחנה השניה והשלישית מבטיחות שהם לא משפיעים על הערך הסופי. מצד שלישי, האבחנה הרביעית מלמדת על תרחיש אפשרי של שגיאת זמן ריצה או לולאה אינסופית בביטוי שאינו אחרון שלא תתרחש בתוכנית ה python המומרת.

לסיכום:
תוכנית ה python אינה שקולה אם ביטוי בגוף הפרוצדורה שאינו האחרון כולל לולאה אינסופית או שגיאת ריצה.
בכל מקרה אחר תוכנית ה python תהיה שקולה.

מפתח ניקוד:

- מי שהצביע על הבעייתיות שבתרגום הביטוי האחרון בלבד (האבחנה הראשונה), אך לא התייחס לכך שב-L2 חוזר הערך של הביטוי האחרון בלבד (האבחנה השניה) או שאין side effects (האבחנה השלישית) וגם לא ציין את אפשרות השגיאה או הלולאה האינסופית (האבחנה הרביעית): ירדו שתי נקודות.
- מי שציין שהתוכנית שקולה למרות שמחזירים רק את הביטוי האחרון (האבחנה הראשונה) כיוון שזו סמנטיקת הפעלת פרוצדורה ב L2 (האבחנה השניה), אך לא התייחס לעניין ה side effects: ירדה נקודה אחת.
- ירדו 4 נקודות למי שטען שכל הביטויים בפרוצדורה מתורגמים.
- 0 נקודות למי שציין דברים שאינם קשורים ואינם נכונים.

ב.2 [3 נקודות]

```
f = lambda p : p[0] + p[1]
f((1,2))
```

ב.3 [10 נקודות]

```
export const l30ToPython = (exp: Parsed | Error): string | Error =>
    isError(exp) ? exp.message :
    isProgram(exp) ? map(l30ToPython,exp.exps).join("\n") :
    isBoolExp(exp) ? (exp.val ? 'True' : 'False') :
    isNumExp(exp) ? exp.val.toString() :
    isVarRef(exp) ? exp.var :

    isLitExp(exp) ? "()" :

    isDefineExp(exp) ? exp.var.var + " = " + l30ToPython(exp.val) :
    isProcExp(exp) ? "(" + "lambda " +
                         map((p) => p.var, exp.args).join(",") + ": " +
                         l30ToPython(exp.body[exp.body.length-1]) +
                     ")" :
    isIfExp(exp) ? "(" + l30ToPython(exp.then) +
                       " if " +
                       l30ToPython(exp.test) +
                       " else " +
                       l30ToPython(exp.alt) +
                   ")" :
    isAppExp(exp) ?
        (isPrimOp(exp.rator) ?
            primOpApp2Python(exp.rator, exp.rands) :
```

```
            l30ToPython(exp.rator) + "(" +
                        map(l30ToPython, exp.rands).join(",") + ")") :
    Error("Unknown expression: " + exp.tag);


const primOpApp2Python = (rator : PrimOp, rands : CExp[]) : string =>
    rator.op === "not" ? "(not " + l30ToPython(rands[0]) + ")" :
    rator.op === "and" ? "(" + map(l30ToPython,rands).join(" && ") + ")" :
    rator.op === "or" ? "(" + map(l30ToPython,rands).join(" || ") + ")" :

    rator.op === "cons" ? "(" + l30ToPython(rands[0]) + "," +
                                l30ToPython(rands[1]) + ")" :
    rator.op === "pair?" ? "isinstance(" +
                                   l30ToPython(rands[0]) + ", tuple)" :
    rator.op === "car" ? l30ToPython(rands[0]) + "[0]" :
    rator.op === "cdr" ? l30ToPython(rands[0]) + "[1]" :

    "(" + map(l30ToPython,rands).join(" " +
                        (rator.op === '=' ? '==' : rator.op) + " ") + ")"
```

שאלה 4: אינטרפרטר של מודל ההצבה/החלפה [20 נקודות]

א. Applicative Order מול Normal Order

1.א

ההבדל בין שני האלגוריתמים נעוץ בשערוך ביטוי הפעלה של פרוצדורת משתמש. אינטרפרטר הפועל לפי applicative order יפעל באופן רקורסיבי על האופרנדים / ארגומנטים המעורבים בהפעלה ויחשב ערכם. ערכים אלו יחליפו את מופעי המשתנים המתאימים בגוף הפרוצדורה לשם המשך פעולת החישוב. במילים אחרות, סדר הפעולות באינטרפרטר כזה הוא eval-substitute-reduce. לעומת זאת, סדר הפעולות באינטרפרטר הפועל לפי normal order הוא substitute-reduce: מופעי המשתנים המתאימים בגוף הפרוצדורה מוחלפים בביטויי האופרנדים, לא בערכם, וגוף הפרוצדורה מחושב מיד. הביטויים יחושבו רק כאשר ערכם יידרש על מנת להחליט כיצד להמשיך בשערוך (לדוג' כאשר יש לחשב את ה - test בביטוי תנאי) או כאשר הביטוי הוא ארגומנט להפעלת פרוצדורה פרימיטיבית.

2.א

```
(define loop
     (lambda() (loop)))

((lambda(x) 'ok)
 (loop))
```

הערה: דוגמאות בהן החישוב הסתיים בשגיאה במקום להמשיך בלולאה אינסופית במקרה של applicative-eval התקבלו, אך ניתן עליהן ניקוד חלקי.

3.א

נעדיף להשתמש באלגוריתם normal-eval כאשר ערכי האופרנדים לא יהיו בהכרח נחוצים לחישוב. כך, נימנע מחישוב מיותר. לדוג':

```
(define long-comp (lambda(x) // perform a very long computation))
(define prob-foo
     (lambda(x y)
          (if (< x 0.5) y 0)))
(prob-foo 0 (long-comp 1))
```

נעדיף להשתמש באלגוריתם app-eval כאשר ערכו של אופרנד משמש לעיתים קרובות בתהליך החישוב. באופן זה, נמנע מחישוב חוזר של אותו הביטוי. לדוג':

```
(define cond-foo (lambda(x) (cond ((= x 1) (do something…))
                                  ((= x 2) (do something else…))
                                  :
                                 (else… ))))
(cond-foo (long_comp 0))
```

ב. ערכים וביטויים ליטרליים

ב.1

The applyProc procedure receives arguments which are all of type Value ,proc is a Value which can be either a PrimOp or a Closure value, rands is a list of Values

The body of the closure is a list of CExp expressions. Our objective is to replace all VarRef occurrences in the body with the corresponding values of the arguments (in our example, we want to .(replace (VarRef x) with 5

There is a typing problem with this operation: 5 is a Value, while (VarRef x) is an expression. If we .replace (VarRef x) with the value 5 (a number), the resulting body is not a valid AST

To address this discrepancy, we must map the values of the arguments to corresponding .expressions. This mapping is performed in our interpreter with the ValueToLitExp function

ב.2

In the Normal-Eval it does not evaluate expressions' value unless it returns the value, so when passing arguments to a closure it delivers the expressions and not the values (unlike the applicative evaluation algorithm).