מבחן בקורס: עקרונות שפות תכנות, 202-1-2051
מועד: ב
תאריך: 19/7/2023
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]

הערה: בשאלה זו נתייחס בין היתר למימוש של מודל הסביבות עם normal-order.
במימוש זה:

- ה-bindings בפריימים של הפעלות הקלוז'רים קושרים שם משתנה לביטוי, במקום לערך כמו ב-applicative order שמומש בכיתה.
- הפונקציה applyEnv מחשבת את הביטוי המקושר לשם המשתנה הנתון ומחזירה את ערכו, במקום רק להחזיר את ערכו הנתון בפריים כמו ב-applicative order.
- הסביבה הגלובלית נשארת כשהייתה: ה-bindings קושרים שמות משתנים לערכים, ו-applyEnv מחזירה את הערך המקושר למשתנה נתון.

כזכור, הפונקציה and מקבלת שני פרמטרים ומחזירה true אם הערך של שניהם true, אחרת היא מחזירה false.
באינטרפרטר שכתבנו (סוג השפה, L1-L4, אינו רלבנטי לשאלה זו) מומשה הפונקציה and כאופרטור פרימיטיבי.

בסמנטיקת ה-shortcut של פעולה זו, אם ערכו של הפרמטר הראשון הוא false, הפרמטר השני אינו מחושב (כי אין בכך צורך) והפונקציה מחזירה מיד false.

א. עבור כל אחד מהאינטרפרטרים הבאים, ציינו האם סמנטיקת ה-shortcut מתקיימת בפועל על הפעלת and כאופרטור פרימיטיבי:

- מודל ההצבה/ההחלפה, applicative order (במימוש שנלמד בכיתה)

לא. באופרטור פרימיטיבי הפרמטרים מחושבים לפני ההפעלה.

- מודל ההצבה/ההחלפה, normal order (במימוש שנלמד בכיתה)

לא. באופרטור פרימיטיבי הפרמטרים מחושבים לפני ההפעלה, גם ב-normal order.

- מודל הסביבות, applicative order (במימוש שנלמד בכיתה)

לא. באופרטור פרימיטיבי הפרמטרים מחושבים לפני ההפעלה.

- מודל הסביבות, normal order (במימוש המתואר בהערה למעלה)

לא. באופרטור פרימיטיבי הפרמטרים מחושבים לפני ההפעלה, גם ב-normal order.

[8 נקודות]

ב. ממשו את and כפרוצדורת משתמש (בשם my_and)

```
(define my-and
   (lambda (a b)
     if a b #f
  )
)
```

[3 נקודות]

ג. תארו את החישוב של התוכנית הבאה בכל אחד מהאינטרפרטרים הבאים, וציינו עם נימוק קצר האם מתקיימת סמנטיקת ה-shortcut עבורו.

בתיאור החישוב עבור מודל ההצבה/ההחלפה, ציינו את הביטוי הנשלח בכל שלב לפונקציית ה-eval.
לדוגמא:

עבור

```
(define square (lambda (x) (* x x))
(square 3)
```

יש לכתוב

```
eval: square
eval: 3
eval: (* 3 3)
```

בתיאור החישוב עבור מודל הסביבות, ציירו את דיאגרמת הסביבות.

```
(define x 1)
(define f
  (lambda (a) (> a x))

(my-and #f (f 2))
```

- מודל ההצבה/ההחלפה, applicative order (במימוש שנלמד בכיתה)

```
eval:(my-and #f (f 2))
eval: my-and
eval: #f
eval: (f 2)
eval: f
eval: 2
eval: (> 2 x)
```

```
eval: >
eval: 2
eval: x
eval: (if #f #t #f)
```

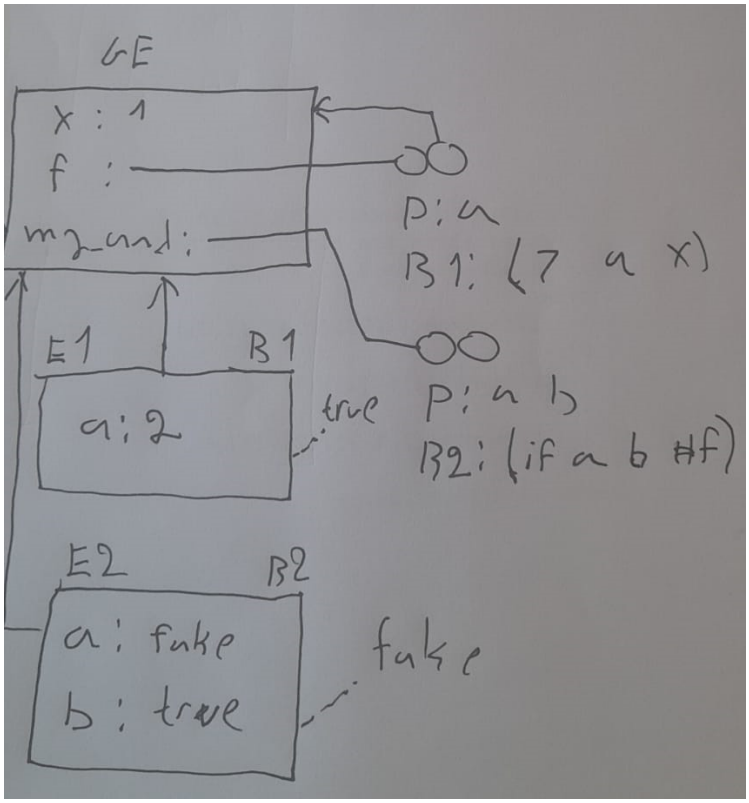
האם מתקיימת סמנטיקת ה-shortcut: **לא, (f 2) חושב בכל מקרה**

- מודל ההצבה/ההחלפה, normal order (במימוש שנלמד בכיתה)

```
eval:(my-and #f (f 2))
eval: my-and
eval: (if #f (f 2) #f)
eval: #f
```

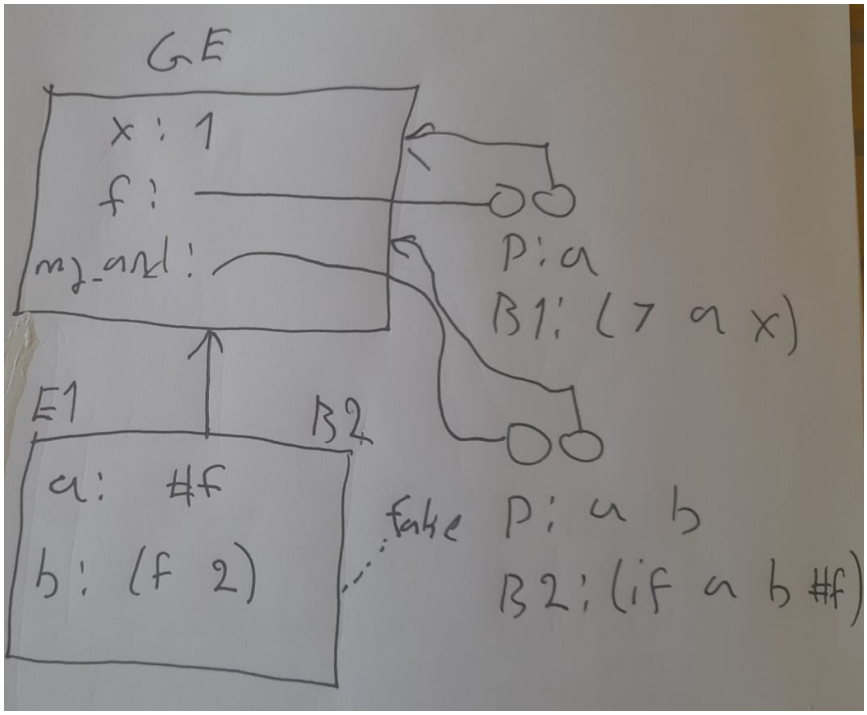
האם מתקיימת סמנטיקת ה-shortcut: **כן, (f 2) לא חושב**

- מודל הסביבות, applicative order (במימוש שנלמד בכיתה)



האם מתקיימת סמנטיקת ה-shortcut: **לא, (f 2) חושב בכל מקרה**

- מודל הסביבות, normal order (במימוש המתואר בהערה למעלה)



האם מתקיימת סמנטיקת ה-shortcut: **כן, (f 2) לא חושב**

[16 נקודות]

ד. כמסקנה מסעיפים א-ג, האם הייתם ממליצים לממש את and כצורה מיוחדת? אם כן, תארו במילים את הסמנטיקה של צורה זו. אם לא, הסבירו בקצרה מדוע זה לא נדרש.
[3 נקודות]

אם רוצים לחסוך בחישובים ובו בזמן לא להיות תלויים במדיניות של סדר חישוב מסוים, הייתי מרחיב את השפה עם צורה מיוחדת חדשה. כצורה מיוחדת, ניתן לממש את and כך שקודם מחושב הארגומנט הראשון והארגומנט השני מחושב רק אם ערכו true.

ה. כדי להימנע קטגורית מבעיית מימוש סמנטיקת ה-shortcut בכל תצורה של האינטרפרטר, הציעה אחת הסטודנטיות לממש את my_and בסמנטיקת ה-shortcut תוך שימוש בחישוב עצל עבור הפרמטר השני.
ממשו מחדש את my-and, והדגימו כיצד יש להפעיל אותו על הביטוי מהתוכנית בסעיף ג:

```
(define my-and
  (lambda (a b)
    if a (b) #f
  )
)
```

~~(my-and #f (f 2))~~

```
(my-and #f (lambda () (f 2)))
```

[5 נקודות]

שאלה 2: טיפוסים [20 נקודות]

בשאלה זו נתייחס לשפה L52 שמבוססת על השפה L5 - שפה עם lambda, if, let, letrec, cons, car, cdr ועם type annotations לפי מערכת הטיפוסים הבאה:

```
<TExp> ::= <atomicTExp> | <compoundTExp> | <TVar>
<atomicTExp> ::= boolean | number | string | void
                 | any | never
<compoundTExp> ::= <procTExp>
                 | <typePredTExp>
                 | (union <TExp> <TExp>)
                 | (inter <TExp> <TExp>)
                 | (diff <TExp> <TExp>)
<procTExp> ::= ( <TExp>+ -> <TExp> ) | ( Empty -> <TExp> )
<typePredTExp> ::= ( <TExp> -> is <TExp> )
```

התוספות ב-L52 מעל L5 הן:

1. **any**: this type describes the set of all possible values (כל הערכים)
2. **never**: this type describes the empty set (קבוצה ריקה)
3. **(union <t1> <t2>)**: this type describes the set containing the union of the values in <t1> and <t2> (same as in HW4 in language L51). (איחוד)
4. **(inter <t1> <t2>)**: this type describes the set containing the intersection of the values in <t1> and <t2> (חיתוך)
5. **(diff <t1> <t2>)**: this type describes the set containing the values in <t1> that are not in <t2> (set difference) (הפרש בין קבוצות)
6. **(any -> is number)**: type predicates which are functions of one parameter that return a boolean value and inform the type checker that if the value is true, the parameter belongs to the type, else that the value does not belong to the type.

ה-type predicate מתנהג כמו type predicates ב-TypeScript. למשל:

```
(define (isNumber : (any -> is number))
  (lambda ((x : any)) : is number (number? x)))

Is semantically equivalent to the TypeScript function:

const isNumber = (x: any): x is number => typeof x === "number";
```

2.1 בנאי הטייפים union, inter, diff מוגדרים לפי חוקים של תורת הקבוצות. לכל ביטוי TExp הבא, רשמו ביטוי TExp יותר פשוט המכיל אותה קבוצה של ערכים (**רשמו בתחביר המדויק של Texp**)
[3 נק]

(inter number boolean) _____**never**_____

(inter never string) _____**never**_____

(union never number) _____**number**_____

(diff (union number string) string) _____**number**_____

(inter number (union number boolean)) _____**number**_____

(inter (union boolean number) (union boolean string)) _____

_____**boolean**_____

2.2 השלימו את הקוד הבא כך שהוא יעבור type checking ב-L52 אך לא עובר ב-L51
[3 נק]

```
;; Return type (any -> is number) defined in L52
;; In L51 the return type would be boolean
(define (isNumber : (any -> is number))
  (lambda ((x : any)) : is number
    (number? x)))

;; Function to complete
(define (good_in_L52 : ((union number boolean) -> number))
    (lambda ((x : (union number boolean))) : number
        (if (isNumber x)
            (* x 2)
            0
    ))
```

This procedure does not pass type checking in L51 because the expression (* x 2) is not safe when x is known to be (union number boolean) - but in L52, the expression is guarded by the type predicate (isNumber x) - and hence the type checker accepts it as safe.

2.3 עם ההגדרה של union ו-inter - הגדרנו את יחס הסדר החלקי isSubType בין ביטויי TExp.
הרחיבו את הפונקציה שהוגדרה ב-HW4 כדי לכסות את המקרים של any ו-never.
(אין צורך לטפל במקרים עם inter או diff)
[3 נק]

```
// Add cases for comparison with never and any
const isSubType = (te1: TExp, te2: TExp): boolean =>
    (isTVar(te1) && isTVar(te2)) ? equals(te1, te2) :
    isTVar(te1) ? true :
    isTVar(te2) ? true :

    // cases with never or any
    isNeverTExp(te1) ? true :
    isAnyTExp(te2) ? true :
    isNeverTExp(te2) ? false :
    isAnyTExp(te1) ? false :

    (isUnionTExp(te1) && isUnionTExp(te2)) ? isSubset(te1.components, te2.components) :
    isUnionTExp(te2) ? containsType(te2.components, te1) :
    (isProcTExp(te1) && isProcTExp(te2)) ? checkProcTExps(te1, te2) :
    isAtomicTExp(te1) ? equals(te1, te2) :
    false;
```

2.4 השלימו את ה-return type של פונקצית f ב-L52 - אם היא לא עוברת type checking רשמו Failure - אם היא כן עוברת type checking רשמו את הביטוי TExp של ה-return type בתחביר תקין של L52 (תחביר לא תקין לא יתקבל). נמקו.
[3 נק]

```
(define (is_number? : (any -> is number))
    (lambda ((x : any)) : is number
        (number? x)))

(define (is_boolean? : (any -> is boolean))
    (lambda ((x : any)) : is boolean
        (boolean? x)))

(define f
    (lambda ((x : (union number boolean))) : (union number string)
        (if (is_number? x)
            (if (> x 0)          ←— this is safe because of the type predicate above
                "positive"
                "negative")
            (if (is_boolean? x)
                1
                x))))        ← after testing false for (is_number? x) and (is_boolean? x)
                               We know that x must be never
```

נימוק:

1. **When entering the procedure body - we assume that x has type (union number boolean) (by ProcExp typing rule).**
2. **When analyzing the "then" part of the external IfExp with test (is_number? x) - we assume that x has type (inter (union number boolean) number) = number. Therefore, (> x 0) is safe. The "then" part returns a type = (union string string) = string.**
3. **When analyzing the "else" part of the external IfExp - we assume that x has type (diff (union number boolean) number) = boolean.**
4. **When analyzing the "then" part of the inner IfExp (is_boolean? x) - we assume x has type (inter boolean boolean) = boolean.**
5. **When analyzing the "else" part of the inner IfExp - we assume x has type (diff boolean boolean) = never.**
6. **Therefore, the return type of the whole inner IfExp is (union number never) = number.**
7. **In total - the return type of the function is (union string number).**

2.5 ה-typing rule של ProcExp ו-IfExp ב-L51 (השפה של HW4 שתומכת ב-union) הוא:
[6 נק]

```
// L51 ProcExp Typing rule:
For any expression Proc = (lambda ((x1 : t1) … (xn : tn)) : returnTE body)
and any type env tenv and any TExp t1, …, tn, returnTE

If   typeof(body, extend-tenv(x1=t1,...,xn=tn; tenv)) = returnTE
Then typeof<(lambda ((x1:t1) … (xn:tn)) : returnTE body), tenv> = [t1 * ... * tn -> returnTE]

// L51 IfExp Typing rule:
For any expression IfExp = (if test then else)
and any type env tenv and any TExp t1 and t2:

if typeof(test, tenv) = boolean
   typeof(then, tenv) = t1
   typeof(else, tenv) = t2
then typeof<(if test then else), tenv> = union(t1, t2)
```

הרחיבו את ה-typing rule ל-L52 עבור ביטויים מהסוג הבא: `(if (typePred var) then else)`
כאשר typePred הוא ביטוי עם type שהוא type predicate ו-var הוא ביטוי מסוג varRef.
ניתן להשתמש ב-union, inter ו-diff:

```
// L52 IfPredType Typing rule:
For any expression IfExp = (if test then else)
and any type env tenv and any TExp tv, t, t1 and t2:

if  test = (typePred var) and
    typeof(var, tenv) = tv and
    typeof(typePred, tenv) = TypePredTExp(tv -> is t) and

    typeof(then, extend-tenv(var = inter(tv, t); tenv) = t1 and
    typeof(else, extend-tenv(var = diff(tv, t); tenv) = t2

Then typeof< (if (typePred var) then else), tenv> = union(t1, t2)
```

NOTES: The effect of using a type predicate in the `<test>` component of the `IfExp` is that the type checker can rely on the result of the type predicate when analyzing the `<then>` and `<else>` components. This is expressed in the typing rule by extending the `TEnv` in which `<then>` and `<else>` are analyzed: `<then>` is evaluated when the type predicate is true, in this case, we know that the variable `var` has type `t`; this is in addition to the fact that `var` was declared as a variable of type `tv`. We encode this in `extend-tenv(var = inter(tv, t), tenv)`.
When the type predicate is false, we know that `var` is **not** of type `t` - while we still know that it is of type `tv`. We encode this in `extend-tenv(var = diff(tv, t), tenv)`.
Pay attention that the <then> and <else> components are analyzed in different TEnvs - this pattern is fundamental in all the code of the interpreters we have written over the semester. It allows writing code for structural induction with type predicates as guards.

2.6 איזה תכונה של ה-type checker מושפעת במידה ומתכנת כותב type predicate שקרי. למשל:
[2 נק]

```
(define (is_number? : (any -> is number))
    (lambda ((x : any)) : is number
        (string? x)))                    ;; ←- test string? for “is number”
```

הסבר בקצרה עם דוגמה.

The type checker is not sound anymore (נאותות) - that is, it is possible that the type checker decides an expression is type safe - and still, when we execute it we will get a runtime type error. For example:

```
(define (is_number? : (any -> is number))
    (lambda ((x : any)) : is number
        (string? x)))                       ;; ←- test string? for “is number”

(define (f : ((union string number) -> boolean))
    (lambda ((x : (union string number))) : boolean
        (if (is_number? x)
            (> x 0)
            #f)))
(f “a”)  → The type checker decides this is safe - but we fail at runtime on (> x 0)
```

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

3.1 [4 נק']
ציינו שני מאפיינים של פרדיגמת התכנות הפונקציונאלי
א. משתנים אינם משנים את ערכם (immutability). תכונה זו מקלה על הוכחת נכונות של תוכניות.
ב. פונקציות הן ערך כמו כל ערך אחר. אפשר להעביר פונקציות כפרמטר, ולקבל פונקציה כערך חוזר.
ג. הערכה עצלה. הפרדיגמה תומכת בהערכה עצלה, מה שמאפשר חישוב יעיל יותר במקרים מסויימים.
ד. אין משתנים מקומיים (ניתן לדמות משתנים מקומיים ע"י פרמטרים לפונקציה אנונימית)

טעויות נפוצות:
- **מאפיין שקיים בשפות רבות (למשל היכולת להגדיר פונקציה): 1-**
- **-**

3.1 [4 נק']
ציינו שני תרחישי שימוש (use cases) בהם עדיף שימוש בפרדיגמת **התכנות הפונקציונאלי**:
א. תכנות מקבילי: העובדה שמשתנים אינם משנים את ערכם, וספציפית משתנים משותפים בין תהליכים אינם משנים את ערכם, מקל מאוד על תכנות מקבילי בכך שנמנעים מ-race conditions.
ב. חישובים מתמטיים: פרדיגמה זו נבנתה סביב מושגים מתמטיים, ולכן קל לתרגם נוסחה מתמטית לביטוי.
ג. עיבוד מבוזר: תמיכה בפונקציות מסדר גבוה, כלומר האפשרות לשנע את החישוב, ולא את הנתונים, מאפשרים עיבוד מבוזר בקלות.

3.2 [4 נק']
ציינו שני תרחישי שימוש (use cases) בהם עדיף **להמנע** מפרדיגמת התכנות הפונקציונאלי.
א. מהירות חישוב: תכנות פונקציונלי כרוך ברמות גבוהות של הפשטה ואי-שינוי, מה שיכול להפוך את החישוב למאוד לא יעיל. בתרחישים קריטיים לביצועים כגון יישומים

בזמן אמת, שבהם שליטה ברמה נמוכה ואופטימיזציות עדינות הן חיוניות, פרדיגמות אחרות עשויות להיות מתאימות יותר.
ב. שימוש רב בזיכרון: תכנות פונקציונלי מעודד שימוש רב בזיכרון. ישומים שבהם שימוש בזיכרון הוא קריטי לא מתאימים למימוש בפרדיגמה זו.
ב. עבודה עם חומרה: העובדה שתכנות פונקציונאלי אינו מעודד שינוי state או mutation, מקשה מאוד על עבודה עם חומרה.

3.2 [2 נק']
מהן מונאדות בתכנות פונקציונלי?
תבניות עיצוב (design patterns) שנועדו לאפשר הרכבה של פונקציות עם טיפוסים מורכבים.

3.3 [7 נק']
בסעיפים הבאים נבנה מספר פונקציות עזר על מנת לממש, בעזרת `reduce`, פונקציה אשר מקבלת רשימה וסופרת כמה פעמים מופיע כל איבר בה.

פונקציית העזר הראשונה אותה נממש בסעיף זה נקראת dict-get, והיא תמומש בסגנון CPS. היא מקבלת (1) רשימה, המשמשת על תקן מילון, כך שכל איבר ברשימה הוא זוג: מפתח וערך, (2) מפתח, (3) continuation עבור הצלחה ו-(4) continuation עבור כישלון. אם המפתח (2) נמצא בתוך המילון, אז חוזר הערך המשוייך למפתח. אם המפתח לא נמצא, אז מופעלת פונקציית הכישלון. השלימו את קוד הפונקציה:

```
;; Signature: dict-get(dict key succ fail)
;; Type:
;; [List(Pair(T1,T2)) * T1 * [T2 -> T3] * [Empty -> T4] -> T3 union
T4]
;; Purpose: Returns the value associated with the key.
;; Examples:
;;  (dict-get '((a . 1) (b . 2))) 'a id (λ () #f)) => 1
;;  (dict-get '((a . 1) (b . 2))) 'c id (λ () #f)) => #f
(define dict-get
  (λ (dict key succ fail)
    (if (empty? dict) (fail)
        (let ((curr-key (caar dict))
              (curr-val (cdar dict)))
          (if (eq? key curr-key)
              (succ curr-val)
```

```
            (dict-get
             (cdr dict)
             key
             succ
             fail))))))
```

קיבלנו גם תשובה עם שימוש ב-filter.

3.4 [7 נק']

פונקציה הנוספת שצריך לממש היא פונקציה שמעדכנת את המילון. היא מקבלת את המילון, מפתח וערך. אם המפתח קיים במילון, אזי הערך מתעדכן בערך החדש. אם המפתח לא קיים, אז המפתח והערך החדשים נכנסים למילון.

```
;; Signature: dict-set(dict key value)
;; Type: [List(Pair(T1,T2)) * T1 * T2 -> List(Pair(T1,T2))]
;; Purpose: sets a value for the given key.
;; (dict-set '() 'a 1) => '((a . 1))
;; (dict-set '((a . 1) (b . 2)) 'a 10) => '((a . 10) (b . 2))
(define dict-set
  (lambda (dict key value)
    (if (empty? dict)
        (list (cons key value))
        (let* ((item (car dict))
               (curr-key (car item)))
          (if (eq? curr-key key)
              (cons (cons key value) (cdr dict))
              (cons item (dict-set (cdr dict) key value)))))))
```

טעויות נפוצות:

- לא לעשות cons עם שאר המילון.
- שימוש ב-(lambda () #f). זה שגוי כיוון שהערך המשוייך למפתח יכול להיות #f.

3.5 [7 נק']

עכשיו צריך לחבר את שתי הפונקציות יחדיו ולכתוב את פונקציית ה-reducer. הפונקציה מקבלת מילון ומפתח. אם המפתח קיים במילון יש להעלות את הערך ב-1. אם המפתח לא קיים, יש להוסיף מפתח חדש עם הערך 1.

```
;; Signature: value-counts(lst)
;; Type: [List(T1) -> List(Pair(T1,number))]
;; Purpose: Return counts of unique values.
;; (value-counts '(a b a a b c)) => '((c . 1) (b . 2) (a . 3)))
(define value-counts
  (λ (lst)
```

```
(reduce (λ (dict key)
         (dict-get
          dict
          key
          (lambda (curr-count)
            (dict-set dict key (+ curr-count 1)))
          (lambda ()
            (dict-set dict key 1)))) '() lst)))
```

שאלה 4: תכנות לוגי [20 נקודות]

א. מצאו את ה-MGU של כל אחד מזוגות הביטויים הביטויים (אין צורך לתאר את הדרך). אם לא ניתן לבצע יוניפיקציה ביניהם, ציינו את הסיבה לכך.
[8 נקודות]

```
p([X|Xs],[a|Z],Xs)
p(L,L,[])
```

X = a, Xs = [], Z =[], L = [a]

```
p([X|Xs],[a|Z],Xs)
p(L,L,L)
```

No unification due contradiction L = [X|L)

```
p(g([T]), g(T), g)
p(X, Y, T)
```

X = g([g]), Y = g(g), T = g

```
p(cons(a,cons(b,empty)))
p([a,b])
```

Cannot be resolved: the literal expression of lists in Prolog is based on functors and an empty-list symbol, but we don't know they are *cons* and *empty*.

ב. כזכור, המספר ה-חי בסדרת פיבונאצ'י הוא סכום שני המספרים שלפניו. כאשר שני האיברים הראשונים בסדרה הם 1,1.

ב1. ממשו את הפרוצדורה fib/3, המגדירה את היחס הבא: שלושת המספרים הם שלושה מספרים עוקבים בסדרת פיבונאצ'י.
לייצוג מספרים נשתמש בפנקטור s (ה-Church numbers שנלמדו בכיתה)
[6 נקודות]

```
natural_number(zero).
natural_number(s(X)) :- natural_number(X).


plus(X, zero, X) :- natural_number(X).
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).



%fib/3
fib(s(zero),s(zero),s(s(zero))).
fib(N1,N2,N3):-plus(N1,N2,N3), fib(_N0,N1,N2).



?- fib(s(zero),s(zero),s(s(zero)))
true

?- fib(s(s(zero)),s(zero),N3)
false

?-fib(N1,N2,s(s(zero)))
N1 = s(zero), N2 = s(zero)
```

ב2. ממשו את הפרוצדורה fib/2, המגדירה את היחס הבא: הפרמטר הראשון הוא האינדקס של מספר בסדרת פיבונצאצ'י, והפרמטר השני הוא המספר הזה.
[6 נקודות]

```
%fib/2
fib(zero,s(zero)).
fib(s(zero),s(zero)).
fib(s(s(N)),F):-fib(s(N),F1), fib(N,F2), fib(F2,F1,F).

?- fib(zero, s(zero))   % fib0 = 1
```

```
true

?- fib(s(zero), s(zero))    % fib1 = 1
true

?- fib(s(s(s(s(zero)))), N)
N = s(s(s(s(s(zero)))))     % fib4 = 5
```