.1

.a Polymorphic type expressions - הם ביטויים מופשטים המכלילים טיפוסים וביטויים שונים תחת ביטוי אחד. המתאר למעשה מספר טיפוסים קונקרטיים לדוגמה:
(lambda(x) x) הוא ביטוי פולימורפי כיוון שמתאר את הטיפוסים:
[Number->Number], [[Boolean -> Number] -> [Boolean -> Number]] וכדומה.
מהיותו פולימורפי הוא יסומן כ- T_i לדוגמה בפונקצית הזהות $[T_i \rightarrow T_i]$.

Polymorphic language expressions - הם ביטויים גנריים שאינם מגדיר במפורש את הטיפוס שלו, לדוגמא (cons x y) מחזיר מקבל T1,T2 ומחזיר Pair(T1,T2) שזהו טיפוס גנרי.

.b *value constructor* - בכדי לאתחל משתנה, בנאי הערך מקבל ערך השמה אשר מועבר כארגומנט לבנאי המתאים לטיפוס המשתנה, אם הערך תואם את דרישות ההשמה של המשתנה. ההשמה מתבצעת.

.i Scheme - ניצור משתנה כללי ע"י השורה: $(define\ i\ 42)$.
תתבצע בדיקה עבור 42 לסוג הטיפוס שלו (Number) וכן בהתאמה לסוג הבנאי המתאים, תתבצע השמה של הערך 42 למשתנה i דרך הבנאי [Number -> Void].

.ii Java - ניצור משתנה ע"י השורה: int i.
ברגע שנבצע את השורה i = 42. יקרא בנאי ההשמה של i, אשר יבדוק האם ה type של i יכול לקבל ערכים מספריים (42), אם כן, ישמר בזיכרון הערך 42 למשתנה i.

type constructor - בניגוד לvalue constructor מייצר את סוג הטיפוס של האובייקט.

.i scheme - השפה תומכת ב-polymorphic procedures ושערוך הטיפוס מתבצע ע"י Polymorphic type expressions.

.ii Java - לכל אובייקט יש טיפוס המותאם לו, והבנאי עובד בצורה דומה לvalue constructor.

.c {x:Number, f:[Number -> T]} |- (f x):T - הנוסחה אומרת כי תחת ההנחה שהטיפוס של המשתנה x הוא Number, וההנחה שהפרוצדורה f היא מהטיפוס [Number -> T] אז ההפעלה של הפרוצדורה f על המשתנה x תחזיר לנו טיפוס T פולימורפי.

.d שפה נחשבת fully typed syntax - כאשר כל המבנים והטיפוסים בשפה מוגדרים היטב, כלומר דורשים תאור מדויק של הטיפוסים (כולל משתנים, מבנים ופונקציות). לדוגמא שפות כמו java, c ,c++ וכדומה.
שפה נחשבת partially typed syntax כאשר המבנים בה לא בהכרח מוגדרים היטב, כלומר מקבלים תאור וטיפוס, ומשתמשים בטיפוסים ופרוצדורות פרימיטיביים המובנים בשפה כדי לתאר את את המידע עבור המבנה או הפונקציה. לדוגמה שפות כמו scheme, prolog.

.e Dynamic typing - ,אך ורק תוכניות שהם partially typed syntax יתייחסו לתרגום כלל הפקודות הנכתבות תוך כדי ריצה. וזריקת שגיאה ויציאה מן התוכנית אם ארעה שגיאה סנטקטית. מתודולוגיה זו מתבצעת על משתנים מוגדרים היטב בכדי בחינת הtype tagging.
לדוגמא שפות כגון scheme, prolog.
Static typing - מתייחס לשפות שהם fully typed syntax, בשפות אלו כלל הטיפוסים מוגדרים היטב ומראש ואין בעיה לקבוע את סוג הטיפוס ודרך הפעולה עליו. לדוגמא שפות כגון: C++ .

f. יתרונות:
- מאפשר להפוך כל תהליך רקורסיבי לאיטרטיבי,כלומר כל קריאה רקורסיבית מתבצעת בעמדת זנב.
- מאפשר לשלוט בסדר לפיו התוכנית תחשב ביצועים.
- מהווה כלי לפישוט קוד ארוך ומסובך, נהוג בכתיבת קומפליירים.
- מאפשר להחזיר כמה ערכים יחדיו ידידי.

חסרונות:
- צורך מקום רב של זיכרון, כמספר הקריאות הרקורסיביות.

2.

a. לא, להלן דוגמא נגדית:(עמוד 81):

```
>(define f (lambda (x) (f x)))
>(define g (lambda (x) 5))
>(g (f 0))
```

b. נכון

c. נכון

d. לא, להלן דוגמא נגדית:

```
>(cons 1 'ppl)
>'(1 . ppl)
1 - Number
ppl - Symbol
```

e. נכון

```
>(define x 5)
>(+ x y)
```

f. לא נכון

נתבונן על מימוש מס' 1:

```
> (make-rat(n,d) 10 100)
> '(10 . 100)
>(numer '(10 . 100))
>10
```

נתבונן על מימוש מס' 3:

```
> (make-rat(n,d) 10 100)
> (numer '( 1 10))
> 1
```

מכיוון שמימוש מס' 3 מבצע צמצום של השבר כמה שניתן, אנו לא מקבלים חזרה את ערך n המקורי.

g. נכון,

מימוש lazy procedual inpementaion ממש Visitor pattern, בכך שהאובייקט מקבל selector ומתבצע double displatch בין הADT ולבין הselector ובכך מופעלים operations על האובייקט מבלי לבצע שינויים בערכים הקיימים.

(a (2

```
(atomic? (make-proc-te (make-tuple-te (list 'Number)) ‘Number)) ==> #t
applicative-eval[(atomic? (make-proc-te (make-tuple-te (list 'Number)) ‘Number))]
applicative-eval[atomic?] => #<procedure atomic?>
        applicative-eval[(make-proc-te (make-tuple-te (list 'Number)) ‘Number))]
        applicative-eval[make-proc-te] => #<procedure make-proc-te>
                applicative-eval[make-tuple-te] = #<procedure make-tuple-te>
                        applicative-eval[list ‘Number]
                                applicative-eval[list] = #<primitiv-procedure list>
                                        applicative-eval[‘Number] => #<symbol>
                        =>(Number)
                ⇒ (* Number).

                applicative-eval[‘Number] => #<symbol>
                => ‘Number
        ⇒ ( -> (* Number) Number)
=> #t
(same to the boolean applicative-eval).

(atomic? (make-proc-te (make-tuple-te (list 'Number)) 'T1)) ==> #f
aplicative-eval[(atomic? (make-proc-te (make-tuple-te (list 'Number)) 'T1))]
        aplicative-eval[atomic?] => #<procedure atomic?>
        aplicative-eval[(make-proc-te (make-tuple-te (list 'Number)) 'T1))]
                aplicative-eval[make-proc-te] => #<procedure make-proc-te>
                aplicative-eval[(make-tuple-te (list 'Number))]
                        aplicative-eval[make-tuple-te] => <procedure make-tuple-te>
                        aplicative-eval[(list 'Number)]
                                aplicative-eval[list] => #<primitive-procedure list>
                                aplicative-eval['Number] => Symbol
                        => (Number)
                => (* Number)
                aplicative-eval[‘T1] => Symbol
                => ‘T1
        => (-> (* Number) T1)
=> #f
```

(tuple? (make-proc-te (make-tuple-te (list 'Number)) 'T1)) ==> #f
tuple?(te)

נרצה לבדוק האם הפרמטר te הוא מטיפוס list

(list? (make-proc-te (make-tuple-te (list 'Number)) 'T1))
aplicative-eval[(list? (make-proc-te (make-tuple-te (list 'Number)) 'T1))]
    aplicative-eval[list?]
    aplicative-eval[(make-proc-te (make-tuple-te (list 'Number)) 'T1))]
        aplicative-eval[make-proc-te] => #<procedure make-proc-te>
        aplicative-eval[(make-tuple-te (list 'Number))]
            aplicative-eval[make-tuple-te] => <procedure make-tuple-te>
            aplicative-eval[(list 'Number)]
                aplicative-eval[list] => #<primitive-procedure list>
                aplicative-eval['Number] => Symbol
            => (Number)
        => (* Number)
        aplicative-eval[‘T1] => Symbol

=> ‘T1
    => (-> (* Number) T1)
=> #t

ניתן לראות כי האינווריאנטה של tuple? נשמרת -> מקבלת list כארגומנט.

4.

a. The invatiants:
   - make-sub(variable-list, te-list)
     - make-sub(variable-list, te-list) = u
   - get-variables(sub)
     - get-variables(make-sub(variable-list, te-list))
   - get-tes(sub)
     - get-tes(make-sub(variable-list, te-list)) = List
   - get-expression-of-variable(sub)
     - get-expression-of-variable(make-sub(variable-list, te-list)) =List
   - non-circular?(variable, te)
     - non-circular?
       (memeber (get-variables (make-sub variable-list te-list))
       (member (get-tes (make-sub variable-list te-list)))
   - sub?(sub)
     - sub?(make-sub(variable-list, te-list))
   - empty-sub?(sub)
     - empty-sub?(make-sub(variable-list, te-list))