Question 1: Theoretical Questions

Q1.1:

בתכנות פונקציונלי טהור, פונקציות בעלות ריבוי ביטויים אינו מומלץ וכך גם בשפה L3 שהיא שפה פונקציונלית, בה נוכל להגדיר ביטוי בעזרת let וכך לעשות בו שימוש חוזר. מנגד, בשפות שאינן פונקציונליות נשתמש במספר רב יותר של ביטויים כי אין דרישה לביטויים שמחזירים ערך או לפונקציות שאין להן side-effects כמו בשפות פונקציונליות.

Q1.2:

א. צורות מיוחדות עוזרות להמנע מחזרה מיותרת על קוד קיים, וזה מומלץ מכיוון שחזרה על קוד עלולה להיות פחות קריאה, ובנוסף ליצור שגיאות בהעתקה ולאלץ אותנו לכתוב יותר טסטים. בנוסף, שימוש באופרטורים פרימיטיביים בלבד תגרום לחישוב כפול ומיותר בזמן ריצה.
בשפה L3 נוכל להגדיר צורות מיוחדות כמו בשימוש ב let, וכך לעשות בו שימוש חוזר כמו בדוגמא שניתנה בהרצאה:
בחישוב

```
(+ (/ (sqrt 2) (sqrt 3)) (/ (sqrt 2) (sqrt 3)))
```

בו אנחנו מחשבים את (/ (sqrt 2) (sqrt 3)) פעמיים, נוכל להמנע מחישוב חוזר ע"י הצורה המיוחדת let כך:

```
(let
  (
   (v (/ (sqrt 2) (sqrt 3)))
  )
  (+ v v)
)
```

ב. האופרטור or חייב להיות מוגדר כצורה מיוחד, על מנת שתהיה לנו את האפשרות להשתמש ב shortcut semantics, לדוגמא || או | דורשים התייחסות שונה, כלומר אם True אז לעבור ישר הלאה או לחשב את כל הביטויים הבוליאנים ואז להחליט אם לעבור, בהתאמה. זו התנהגות שונה שאותה נגדיר כפונקציה על פי החלטתינו.

Q1.3:

א. הערך שיוחזר הוא 3. הסבר:

```
(define x 1)
```

פה נגדיר x=1 כמשתנה גלובלי.

```
(let ((x 5)
      (y (* x 3)))
      y)
```

נגדיר משתני סביבה x=5, y=x*3. בקריאה ל y, נחשב x*3 לפי ההשמה x=1 שקרתה לפני ה let הנוכחי, ונקבל 3. זוהי הקריאה האחרונה, לכן יוחזר הערך 3.

ב. הערך שיוחזר הוא 15. הסבר:

```
(define x 1)
```

פה נגדיר x=1 כמשתנה גלובלי.

```
(let* ((x 5)
      (y (* x 3)))
```

```
        y)
```

נגדיר משתני סביבה x=5, y=x*3. בקריאה ל y, נחשב x*3 לפי ההשמה x=5 שקרתה בתוך ה let* הנוכחי, ונקבל 15. זוהי הקריאה האחרונה, לכן יוחזר הערך 15.

ג. נרצה שזמן בחישוב y=x*3, נכיר ב x כ 5. מכיוון שזה לא קורה בתוך ההשמות ב let, נכניס את ההשמה y=x*3 כהשמה ב let השני בגוף ה let הראשון:

```
(define x 1)
(let ((x 5))
        (let ((y (* x 3)))
                y))
```

ד. נגדיר באמצעות lambda:

```
(define x 1)
(lambda (x y) (* x y)) 5 3)
```

Q1.4:

א. תפקיד הפונקציה valueToLitExp הוא למפות את הארגומנטים המשוערכים (Value) לביטויים (CExp) כאשר אנחנו מבצעים applyClosure. פונקציה זו נדרשת משיקולי תאימות טיפוסים.

ב. מכיוון שב Normal Order הארגומנטים לא משוערכים בקריאה ל applyClosure, לכן הם ביטויים ואין צורך לבצע את ההמרה.

ג. מכיוון שגם כאן גוף הפרוצדורה הוא ביטוי והוא נשלח כארגומנט ל applyClosure ולכן אין צורך בהמרה. ביטוי זה ישוערך מאוחר יותר בקריאה evalCExps.

Q1.5:

א. יהיו תוכניות שב Applicative Order נכנס ללולאה אינסופית וב Normal Order לא, וכל ביטוי שעוצר ב Applicative Order יעצור ב Normal Order. לכן מהבחינה הזו, Normal Order היא שפה עשירה יותר.
כמו בדוגמא שנראתה בתרגול 5:

```
(define delta (lambda (x) (x x)))
((lambda (x y) x) (+ 1 1) (delta delta))
```

ב Applicative Order נחשב קודם את (delta delta):
הביטוי הזה שווה ל ((lambda (x) (x x)) delta), כלומר להפעיל על x=delta את (x x). קיבלנו שוב (delta delta) וכך נחזור חלילה ונכנס ללופ אינסופי, בעוד שב Normal Order נעשה (y = (delta delta בלי לחשב את הערך שלו, ואז ב (lambda (x y) x) לא נצטרך את y. כלומר לא נצטרך לעולם לחשב את הערך (delta delta) ולא נכנס ללופ אינסופי.

ב. ה Applicative Order יותר יעיל כשמשתמשים במשתנה יותר מפעם אחת, מכיוון שנוכל להשתמש בתוצאת החישוב שלו, בעוד שב Normal Order נצטרך לחשב אותו כל פעם מחדש. בנוסף, ה Applicative Order יותר אינטואיטיבי ורוב שפות התכנות בנויות על מודל החלפה זה.
כמו בדוגמא שנראתה בתרגול 5:

```
((lambda (x) (+ x x)) (fun 555))
```

כאשר fun זאת פונקציה כלשהי. ב Applicative Order נחשב קודם את ה fun ואז נבצע השמה ל x ואז נבצע כרגיל את החיבור. כלומר חישבנו את fun פעם אחת. מנגד ב Normal Order נחשב את הביטוי הבא:

```
(+ (fun 555) (fun 555))
```

כלומר נחשב את fun פעמיים.

:Q1.6

א. מכיוון שהסביבה הנוכחית או הסביבה שנצביע עליה תכיל את שם המשתנה עם הערך הרלוונטי ל scope הנוכחי, וזה בשונה ממודל ההצבה ב Applicative Order, בו שני משתנים עם הקשר שונה ושם דומה יגרמו לתוצאה שגויה.

ב. במקרה זה renaming לא נדרש במודל ההחלפה מכיוון שהבעיה של Capture נוצרת כאשר יש לנו משתנה חופשי, ובמקרה המתואר אין משתנים חופשיים (כל המשתנים קשורים). כלומר לא נקבל פה התנגשות של שני משתנים עם שם דומה אך עם הקשר שונה.

<u>Question 2d:</u>

convert class exp to proc exp:

```
(define pi 3.14)
(define square (lambda (x) (* x x)))
(define circle
        (lambda (x y radius)
        (lambda (msg)
           (if (eq? msg 'area)
              ((lambda () (* (square radius) pi)))
              (if (eq? msg 'perimeter)
                 ((lambda () (* 2 pi radius)))
                 #f)))))
(define c (circle 0 0 3))
(c 'area)
```

operands to the L3applicativeEval:

3.14
(lambda (x) (* x x))
(lambda (x y radius) (lambda (msg) (if (eq? msg 'area) ((lambda () (* (square radius) pi))) (if (eq? msg 'perimeter) ((lambda () (* 2 pi radius))) #f))))
(circle 0 0 3)
circle
0
0
3
(lambda (msg__1) (if (eq? msg__1 'area) ((lambda () (* (square 3) pi))) (if (eq? msg__1 'perimeter) ((lambda () (* 2 pi 3))) #f)))
(c 'area)
c
'area
(if (eq? 'area 'area) ((lambda () (* (square 3) pi))) (if (eq? 'area 'perimeter) ((lambda () (* 2 pi 3))) #f))
(eq? 'area 'area)
eq?
'area
'area
((lambda () (* (square 3) pi)))
(lambda () (* (square 3) pi))
(* (square 3) pi)
*
(square 3)
square
3
(* 3 3)
*
3
3
pi

environment diagram: