שאלה 1

סעיף א

תפקידו של ה value ADT הוא שsub-eval ידע לזהות ערכים שחושבו בעבר.

בצורה זו נמנע שגיאות שקורות כתוצאה מניסיון לשערך ערכים שכבר שוערכו.

applicative-eval[((lambda (lst)(car lst)) (list 0 1 2))] ==>
Eval:
applicative-eval[(lambda (lst)(car lst))] ==> <Closure (lst)(car lst) >
applicative-eval[(list 0 1 2)] ==> List value ’(0 1 2)’
Substitute: sub[lst,’(0 1 2)’,(car lst)] = (car ’(0 1 2)’)
Reduce:
applicative-eval[(car ’(0 1 2)’)] ==>
Eval:
applicative-eval[car] ==> built-in car code
applicative-eval[’(0 1 2)’] ==> ’(0 1 2)’
==> 0

אם לא היינו משתמשים ב Value ADT היינו מקבלים שגיאה בשורה אחת לפני האחרונה

"0 is not a procedure" .

סעיף ב

ב- env-eval אין צורך ב value ADT.

הערך שחושב נמצא במסגרת של סביבה המתאימה, ואין ניסיון לחישוב נוסף מאחר שמשתמשים בערך שכבר נמצא במסגרת.

בדוגמה הקודמת במסגרת יש התאמה בין lst לבין (2 1 0)' וכך כל שימוש ב lst בסביבה היינו מציבים את (2 1 0)' ללא הוספת ניסיונות חישוב.

סעיף ג

Procedure אינו value אלא ADT שעומד בפני עצמו בגלל שהevaluator הוא זה ששולט ביכולות identify and select, construct .

שאלה 2

חסרונות:

-צריך לעשות התאמות בין ביטויים דומים

יתרונות:

-זמן ריצה קצר יותר

- נותן הפשטה וגמישות לשפה

-לא צריך לרשום מחדש מימושים לפרוצדורות דומות

-הASP אינווריאנטי לכל המימושים לעומת הcore שנכתב מחדש ולכן שמירה על core קטן מקטין את הצורך בכתיבה ארוכה כל פעם מחדש.

שאלה 3

שיפור החלק שמשפיע הכי הרבה על זמן הריצה -חוזרים עליו הכי הרבה פעמים בחישוב-קריאה לפונקציות. בכך נשפר את זמן הריצה של התוכנית.

שאלה 4

מטרתו של המהדר היא ליצור ייעול של פעולת המעריך .במעריך ,ההערכה והניתוח של ביטוים מבוצעים לסירוגין ,לכן אם פונקציה מופעלת כמה פעמים, הביטויים באותה פונקציה ינותחו מספר רב של פעמים .במהדר לעומת זאת, כל ביטוי ינותח פעם אחת בלבד.

דוגמא :

```
(define listlength
  (lambda (list)
    (if
        (null? list)
            0
            (+ 1 (length (cdr list))))))
```

בכל קריאה לפונקציה , length המעריך יצטרך לזהות(מחדש)את ה body -כביטוי if ורק לאחר מכן הוא יוציא מהביטוי את החלקים הרצויים ויחשבם .כדי למנוע חישוב חוזר זה נבצע חישוב חלקי בעזרת Currying .

במקום פונקציה אחת המקבלת שני פרמטרים (ביטוי וסביבה) ,ניצור שתי פונקציות שכל אחת מהן מקבלת פרמטר אחד :המהדר יקבל ביטוי בלבד ויחזיר פונקציה המחכה לקבל סביבה.

המהדר יבצע מראש את ניתוח הקוד ,וכך בהפעלת הפונקציה שהוא מחזיר לא יבוצע ניתוח של הקוד כלל.

שאלה 5

הפונקציה רקורסיבית (ישנם ביטויים שצריכים לגזור מספר פעמים)

יש צורך לבדוק אם הגענו לביטוי ששווה לגזירה שלו , כלומר הגענו לביטוי שלא ניתן לגזירה ולכן הcore יודע לעבוד איתו.

תנאי זה מהווה את תנאי הבסיס של הרקורסיה.

```
(derive (if (> x 0)
            x
            (if (= x 0)
            0
        (- x)))
```

תחילה יהפוך להיות:

```
(cond ((> x 0) x) (else (if (= x 0) 0 (- x))))
```

ובשלב הבא

```
(cond ((> x 0) x) (else (cond ((= x 0) 0) (else (- x)))))
```

ואז נגיע לאותה תוצאה בצורה רקורסיבית ונעצור.

שאלה 6

סעיף א

ערך של פרוצדורה פרימיטיבית הוא לא syntactic expression ולכן לא יכול להיות מחושב שוב, פעולה שנדרשת בחישוב של פרוצדורות משתמש .

לדוגמא :

```
((lambda (foo) (foo (list 0 1))) car)
```