עבודה PPL-3

1)כן, let הינו special form מכיוון שישנו ביטוי שמחושב בצורה שורה מביטוי רגיל, לפי משמעותו של כל ביטוי בתוך הגדרת הlet. כאשר אנו מגדירים ביטוי כזה, אנו יכולים להגדיר משתנים לוקאליים מקושרים לערך שישמשו אותנו בחלק השני שהינו הגוף, ובגוף נקבל כללים לחישוב, שהופכים אותו לצורה מיוחדת, את הlet ניתן בנוסף להמיר ללמבדה.

2)מטרת הפונקציה היא לעטוף את המשתנים שנתונים לנו כערך, ומופיעים בביטוי המתאים לו, למשל אם יש ערך מספרי, נעטוף אותו בnumExp ע"מ שיתאים להחלפות שאנו מבצעים בהערכה של הארגומנטים, ולשמור על מבנה AST תקין. כלומר, כאשר אנו מבצעים הערכה נקבל חזרה value אבל ערכים בגוף הפונק' יהיו מסוג ביטוי ולכן כדי להימנע משגיאה, נעטוף אותם.

3)בשיטה זו, החישוב נדחה לסוף ולא מתבצע ישר, כלומר נדחה את החישוב עד שנגיע לחישוב שאין ברירה אלא לחשבו, ולכן אין ערכים בעץ ה-AST . כל הביטויים מופיעים כcExp בעצמם, ולכן העטיפה כאן מיותרת ולא נדרשת.

4)במודל הסביבות אין לנו צורך בפונק' שתמיר לנו ערכים לביטויים מפני שיש לנו מיפוי של הארגומנטים והערכים להפעלת הפונקציה, ואם נצטרך נחליף את הביטוי לצורך חישוב, בערך ששמור בסביבה ולכן לא מתעוררת הבעיה שקרתה לנו במודל ההחלפה.

5) נרצה לעבור מחישוב אפליקטבי לחישוב נורמלי על מנת למנוע חישובים שלא לצורך, כלומר, יכול להיות לנו ביטוי שלא נהיה צריכים לחשב אותו כלל ונוכל להימנע ממנו אילו רק היינו שומרים את החישוב לסוף ובודקים איזה ביטוי חובה לחשב, לעומת חישוב אפליקטיבי שבהגדרתו מחשב את כל הביטויים. דוג' לכך:

1 : ((3 5 *)(2 1 +)-) ? (x===1) ניתן לראות שבחישוב נורמלי לא היינו מחשבים את הביטוי לאחר סימן השאלה והיינו מחזירים 1 עבור קלט ששונה מ-1 ובחישוב אפליקטיבי בכל מקרה היינו נדרשים לחשב את הביטוי.

6)נרצה לעבור מחישוב נורמלי לאפליקטיבי כאשר נרצה להימנע מחישובים מיותרים, כאשר הביטוי שלנו חוזר מס' רב של פעמים, כאשר נחשב אותו בשיטה האפליקטיבית נימנע מחישובים חוזרים שקורים בחישוב נורמלי. למשל:

(Lambda (x) (* x x) (+(* 5 3)(- 1 2)))

7)א) תחילה, נבחין כי תהליך ההחלפה מחליף הופעות של משתנים חופשיים בביטוי, כאשר החלפה ממפה משתנים לסט של ביטויים ומבוצעת ב2 שלבים: שלב החלפת השם של משתנים והשלב השני הוא ההחלפה בפועל בביטוי המתאים לו, כאשר מטרת ביצוע החלפת שם משתנים היא לטפל בבעיה בה יש לנו משתנים חופשיים בביטוי כולו, כאשר יש לנו שמות דומים להם ולמשתנים קשורים(לארגומנט כלשהו למשל) אשר יכולים ליצור לנו בעיה שבה ניתן ערך שלא רצינו עבור המשתנה החופשי ונקבל תוצאה שגויה. טיפול בבעיה זו מתבצע על ידי החלפת השמות שמבוצעת בשלב הראשון באופרציית ההחלפה. כעת נראה כי על פי הנתונים הקיימים כעת, לא קיימים משתנים חופשיים, כל המשתנים "קשורים" למשתנה אחר כלשהו ולכן לא יכולה לצוץ הבעיה הזו של כפל שמות דומים עבור משתנים קשורים וחופשיים. כלומר, אין כל משמעות לפעולת החלפת השמות, שכן היא מתבצעת באופן כללי על מנת לטפל אך ורק בבעיית כפל שמות עבור משתנים קשורים וחופשיים ולכן עם הנתון שקיבלנו נדע כי המודל הנאיבי נכון גם ללא ההחלפה, כנדרש.

ב) שינוי הביטויים יהיה בכל מקום שבו קיים ב- L3-eval את rename (כלומר נוריד את השימוש בו ונממש את ההחלפה הנאיבית).

1)ב-apply closure:

```
const applyClosure = (proc: Closure, args: Value[], env: Env): Result<Value> => {
    const vars = map((v: VarDecl) => v.var, proc.params);
    const litArgs = map(valueToLitExp, args);
    return evalSequence(substitute(proc.body, vars, litArgs), env);
}
```

2)הקוד עבור substitute נשאר זהה לקוד שאותו כתבנו, פרט לכך שנשמיט כל אזכור עבור renameExps שכן כעת לפי המימוש שלנו אין בו צורך, כמו כן, נוכל להוריד גם את makeVarGen שכן השתמשנו בפונקצייה זו ליצירת שמות משתנים חדשים. ולכן קוד חדש ונקי ל subtitute יהיה:

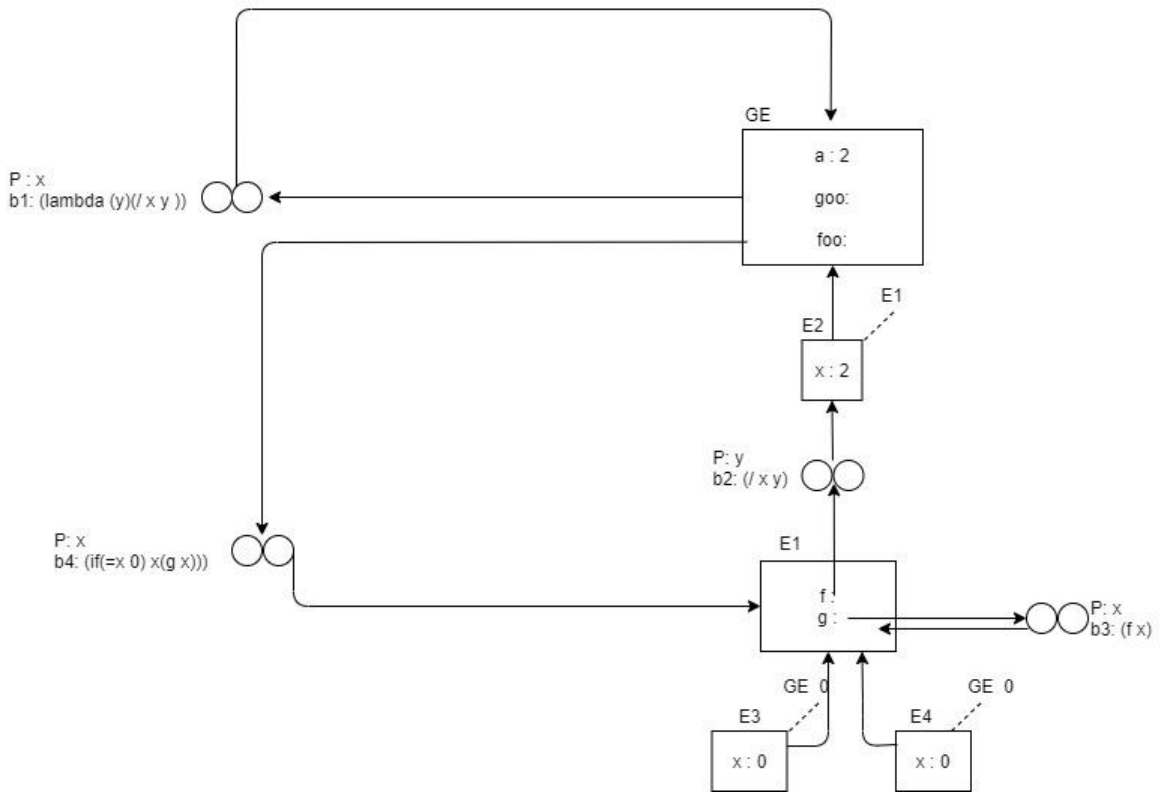
```
export const substitute = (body: CExp[], vars: string[], exps: CExp[]): CExp[] => {
    const subVarRef = (e: VarRef): CExp => {
        const pos = indexOf(e.var, vars);
        return ((pos > -1) ? exps[pos] : e);
    };

    const subProcExp = (e: ProcExp): ProcExp => {
        const argNames = map((x) => x.var, e.args);
        const subst = zip(vars, exps);
        const freeSubst = filter((ve) => !contains(first(ve), argNames), subst);
        return makeProcExp(e.args, substitute(e.body, map((x: KeyValuePair<string, CExp>) => x[0], freeSubst), map((x: KeyValuePair<string, CExp>) => x[1], freeSubst)));
    };

    const sub = (e: CExp): CExp => isNumExp(e) ? e :
        isBoolExp(e) ? e :
        isPrimOp(e) ? e :
        isLitExp(e) ? e :
        isStrExp(e) ? e :
        isVarRef(e) ? subVarRef(e) :
        isIfExp(e) ? makeIfExp(sub(e.test), sub(e.then), sub(e.alt)) :
        isProcExp(e) ? subProcExp(e) :
        isAppExp(e) ? makeAppExp(sub(e.rator), map(sub, e.rands)) :
        e;

    return map(sub, body);
};
```

(8


GE
a : 2
goo:
foo:
P : x
b1: (lambda (y)(/ x y))
E1
E2
x : 2
P: y
b2: (/ x y)
P: x
b4: (if(=x 0) x(g x)))
E1
f :
g :
P: x
b3: (f x)
GE 0
GE 0
E3
E4
x : 0
x : 0