# Assignment 2

## Question 1

1.1

ללא define בשפה L11 נוכל להציג ערכים ללא הגדרתם וללא שם, לכן אין שום שימוש משמעותי בשפה.

1.2

השפה L22 תומכת בפרוצדורות רקורסיביות, שלא ניתן לבטא אם נבטל את define.

1.3

אילו השפה L2 הייתה תומכת באובייקטים מסוג pair, היינו יכולים לשלוח לכל פרוצדורה שדורשת יותר מפרמטר אחד pair או רשימה הבנויה מ - pairs, ולהתאים את הלוגיקה כך שתמומש על פרמטר אחד מסוג pair. אך L2 אינה תומכת בpair ורשימות, לכן לא נוכל לבצע פרוצדורות הדורשות יותר מפרמטר אחד בשפה L22,

לדוגמא: ((lambda (x y) (+ x y)).

1.4

אם נגביל את השימוש בlambda בשפה L23 כך שפונקציה לא תוכל לקבל פונקציה כפרמטר, נאבד את יכולת המימוש של high order function.

Question 2

2.1.a

```
<program> ::= (L3 <exp>+)              / Program(exps:List(exp))
<exp> ::= <define> | <cexp>             / DefExp | CExp
<define> ::= ( define <var> <cexp> )    / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                  / VarRef(var:string)
<cexp> ::= <number>                     / NumExp(val:number)
        |  <boolean>                    / BoolExp(val:boolean)
        |  <string>                     / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(args:VarDecl[],
        |                                /         body:CExp[]))
        |  ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,
        |                                       then: CExp,
        |                                       alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ ) /
LetExp(bindings:Binding[],
        |                                        body:CExp[]))
        |  ( quote <sexp> )                / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )              / AppExp(operator:CExp,
        |                                          operands:CExp[]))
<binding>  ::= ( <var> <cexp> )            / Binding(var:VarDecl,
        |                                           val:Cexp)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
                | cons | car | cdr | list | pair? | list? | number?
                | boolean? | symbol? | string?
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

נבצע עדכון ונוסיף לקטגוריה:

<prim-op> ::= | dict | get | dict?

לקטגוריה <cexp> לא נוסיף נתונים כיוון שהשימוש ב AppExp מספיק

לקטגוריה <sexp> לא נוסיף נתונים כיוון שהוא תומך ברשימות ו pairs , מה שנדרש ליצירת מילון.

2.2.a

```
<program> ::= (L3 <exp>+)              / Program(exps:List(exp))
<exp> ::= <define> | <cexp>             / DefExp | CExp
<define> ::= ( define <var> <cexp> )    / DefExp(var:VarDecl,
val:CExp)
<var> ::= <identifier>                  / VarRef(var:string)
<cexp> ::= <number>                     / NumExp(val:number)
        |  <boolean>                    / BoolExp(val:boolean)
        |  <string>                     / StrExp(val:string)
        |  ( lambda ( <var>* ) <cexp>+ ) / ProcExp(args:VarDecl[],
        |                                /         body:CExp[]))
        |  ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp,
        |                                       then: CExp,
        |                                       alt: CExp)
        |  ( let ( <binding>* ) <cexp>+ ) /
LetExp(bindings:Binding[],
        |                                        body:CExp[]))
        |  ( quote <sexp> )                / LitExp(val:SExp)
        |  ( <cexp> <cexp>* )              / AppExp(operator:CExp,
        |                                          operands:CExp[]))
<binding>  ::= ( <var> <cexp> )            / Binding(var:VarDecl,
        |                                           val:Cexp)
<prim-op>  ::= + | - | * | / | < | > | = | not |  eq? | string=?
                | cons | car | cdr | list | pair? | list? | number?
                | boolean? | symbol? | string?
<num-exp>  ::= a number token
<bool-exp> ::= #t | #f
<str-exp>  ::= "tokens*"
<var-ref>  ::= an identifier token
<var-decl> ::= an identifier token
<sexp>     ::= symbol | number | bool | string | ( <sexp>* )
```

נבצע עדכון ונוסיף לקטגוריה:

<cexp> :: = (dict (<symbol> <cexp>)*)  / DictExp(pairs: [string, CExp][])

2.4

א. נצטרך לשנות רק בהגדרת פרוצדורה. ב- normal order, הפרמטרים לא מוערכים באופן אוטומטי, לכן עלולה להיות השפעה על מתי המילון יבנה. בסעיפים אחרים אין צורך לשנות בגלל שהערכים הם כבר ליטרליים או מפורשים לפי הפרסר.

ב. נצטרך לשנות רק בהגדרת פרוצדורה. הפונקציה משתמשת בסביבה ליצירת קלוז'רים, לכן הערכים בתוך המילון תלויים בהערכת המשתנים בסביבה. בסעיפים אחרים אין צורך לשנות בגלל שכמשתנה פרימיטיבי מתקבל ערך מיידי, וכ- special form מתבצע טיפול בפרמטרים ישירות.

ג. זה לא אפשרי בגלל הפרסר, הוא יראה את זה כאילו a,b הן קריאות לפונקציה עם פרמטרים.

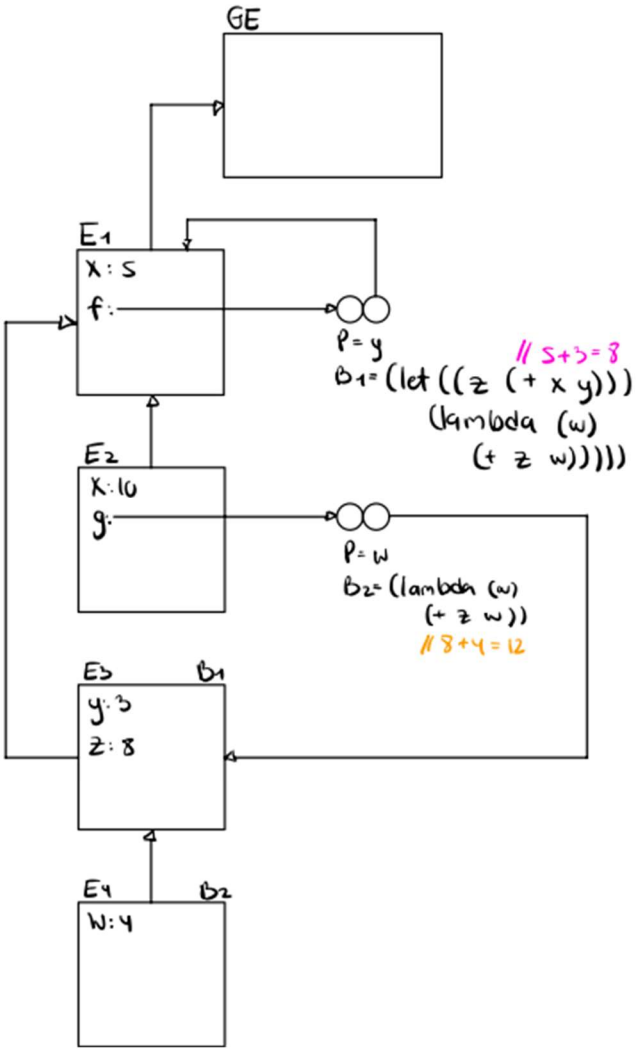
ד. כן, ב2.2 ניתן לרשום: (dict (x (+5 2)) (y (* 2 2))), כך שהערכים של המפתחות מוערכים בזמן ריצה.

לגבי L32, יש ביטויים שלא ניתן להמיר, כמו לדוגמא בעת קבלת מילון שצריך לבצע בו הערכה על הפרמטרים, לא נוכל לבצע הערכה דינמית של הפרמטרים. אי אפשר שהערכים במילון יהיו ביטויים שמחשבים משהו, אלא רק פרמטרים סטטיים.

ה. ההעדפה שלנו היא primitive, הוא פשוט למימוש וקל לבדוק אותו. החסרונות הם שהוא לא גמיש, ולא מאפשר ביטויים דינאמיים.

Question 4

א.




```
(let* ((x 5)
       (f (lambda (y)
            (let ((z (+ x y)))
              (lambda (w)
                (+ z w))))))
  (let ((x 10)
        (g (f 3)))
    (g 4)))
```

יוצר E1 עם x:5
ושם f closure

יוצר E2 עם
x:10
f של הפעלה - g closure

הפעלה פונ', יוצרת E4,
מאפס את g עם w:4

הפעלה פונ', יוצר את E3
ומאפס את f עם פרמטר y:3

הערך הסופי הינו 12.

ב.

```
(let ((x 2)
    (h (lambda (x)
        (let ((f (lambda (y)) (* x y))))
            (f 2)))))
  (h 3))
```