שאלה 1

1. אין תוכנית ב־L1 שאי אפשר להמיר לשפהL11. define רק נותן שם לביטוי, ואפשר תמיד להחליף את השם בביטוי עצמו.
2. קיימות תוכניות ב־L2 שלא ניתנות להמרה ל־L21, למשל פונקציות רקורסיביות, כי define נחוץ כדי לאפשר לקרוא לפונקציה עצמה.
לדוגמה בL2 ניתן לכתוב פונקציה שמחשבת עצרת לכל מספר טבעי ובשפה L21 לא ניתן. כן ניתן לכתוב פונקציה שמחשבת עצרת למספר ספציפי אך לא פונקציה כללית שתעבוד לכל מספר.

```
(define fact
  (lambda n
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

כמו כן, גם את הבעיה הזו ניתן לפתור על ידי שליחת הפונקציה עצמה כפרמטר לפונקציה

3. אין תוכנית ב־L2 שלא ניתנת להמרה ל־L22. L2 בכל מקרה מוגבלת לביטוי אחד וכל פונקציה מרובת פרמטרים ניתן לפרק לשרשרת של פונקציות חד־פרמטריות עם גוף של ביטוי אחד.
לדוגמה פונקציה שמקבלת שני פרמטרים תהפוך לפונקציה שמקבלת את הפרמטר הראשון ומחזירה פונקציה שמקבלת את הפרמטר השני וכך ניתן יהיה לבצע את אותה הפעולה ע״י העברת שני פרמטרים
לדוגמה

```
(lambda (x y) (* x y))
```

הופך ל..

```
(lambda (x) (lambda (y) (* x y)))
```

4. אי אפשר תמיד להמיר תוכנית מ־L2 ל־L23.
אם פונקציה מקבלת פונקציה אחרת כפרמטר (higher-order function), לא תמיד ניתן ליישם זאת ב־L23.

דוגמה:

```
(define apply-twice
  (lambda (f x)
    (f (f x))))
```

apply-twice מקבלת פונקציה f וערך x, ומפעילה פעמיים את f.
אם יודעים מראש את תוכן הפונקציה אפשר לשכתב זאת מבלי להעביר אותה אבל לא ניתן ליצור פונקציה כללית שמקבלת פונקציה ומפעילה אותה פעמיים

שאלה 2

(1

<prim-op> ::= + | - | * | / | < | > | = | not | eq? | string=?
| cons | car | cdr | list | **dict | dict? | get** |pair? | list? | number?
| boolean? | symbol? | string?

(2

```
<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[]))
     | (dict (symbol <cexp>)+)    /DictExp(entries: Entry[])
```

(4

a. במימוש עם primitive operators אין שינוי נדרש. פרימיטיבים כמו dict, get הם מוכללים מראש, והם כן מעריכים את הארגומנטים מיד – תמיד – גם ב־normal order. ולכן ההתנהגות זהה

במימוש (special form), נדרש שינוי אם רוצים שתוכנית תפעל בהתאם ל־**normal order**. בקוד הנוכחי, כל ערכי המילון מוערכים מיידית בעת יצירת המילון עוד לפני שנעשה בהם שימוש. כתוצאה מכך, גם ערכים שלא ניגשים אליהם בפועל — מחושבים מראש, ויכולים לגרום לשגיאות מיותרות. כדי לתמוך ב־normal order, יש לשנות את מימוש evalDictכך שיאחסן את הערכים בצורת Exp, ולהעריך כל ערך רק כשמתבצעת גישה אליו בפועל באמצעות get. שינוי זה יאפשר לדחות את ההערכה עד לרגע שבו יש בה צורך ממשי, כפי שמקובל בהערכה עצלה.

```
const evalDict = (exp: DictExp, env: Env): Result<DictValue> =>
 mapv(
  mapResult((entry) =>
   bind(L32applicativeEval(entry.value, env), (value) =>
    makeOk([entry.key, value] as [SymbolSExp, Value])),
  exp.entries
```

```
),
(entries) => makeDictValue(entries)
);
```

כל הפונקציות ב־2.3 כמו get, dict?, bind וכו' הן טהורות (pure functions), ולכן הסדר שבו הערכים מחושבים (לפני או תוך כדי) לא משפיע על התוצאה הסופית. לכן לא צריך לשנות את הפונקציות, כי אין הבדל בהתנהגות ביניהן במעבר מ־applicative ל־normal order כל עוד אין side effects.

b. ב 2.1 אין הפעלה של פונקציה ולכן לא רלוונטי האם אנחנו בשיטת ההצבה או בשיטת הסביבות, ולכן לא נדרש שינוי בקוד.

ב 2.2 מכיוון ש dict הוא בעצם special form אז נקבל את ההתאמות למודל הסביבות כשנממש את ההתאמות עבור lambda, ולכן אין צורך בהתאמות נוספות.

ב 2.3 כתיבה של פונקציות ב L3 מותאמות למודל הסביבות ומכיוון שזה פונקציות משתמש רגילות ללא שינוי של האינטרפטר לא נצטרך לעשות התאמות.

c. ב־2.2, dict הוא special form – וה־parser מזהה את (dict (a 1) (b 2)) כביטוי מילון (DictExp). המפתחות a, b נשמרים כסמלים (SymbolSExp) מבלי שיערכו, וה־values נשלחים להערכה רק בשלב ה־interpreter. כך ניתן לכתוב את התחביר בצורה טבעית ולוגית.

לעומת זאת, ב־2.1 (פרימיטיב) וב־2.3 (פונקציית משתמש), ה־parser מתרגם את (dict ...) כ־AppExp רגיל — כלומר dict הוא פונקציה שמופעלת עם ארגומנטים.

בשלב ההערכה באינטרפטר:

- ב־applicative order, כל הארגומנטים מוערכים מיידית — ולכן (a 1) מנסה להפעיל את a, וגורם לשגיאה.
- ב־normal order, הארגומנטים מועברים בלי הערכה מיידית, אך עדיין כאילו הם ביטויי קריאה (AppExp) — שגם זה שגוי ולא כפי שהיינו רוצים לשמור את המילון

לכן, ב־2.1 ו־2.3 נדרש להשתמש בציטוט כמו של רשימה על מנת שזה יפוענח כצורת רשימה ולא כהפעלה של פונקציה (המפתח) עם פרמטר שהוא הערך

d. ב־2.2 (special form), ניתן להכניס ביטויים מחושבים כשדות מילון (dict (a (+ 1 2))) אם נעשה זאת ב 2.1 או 2.3 כך: (dict '((a . (+ 1 2)))) כאן הערך לא מוערך — נקבל את הביטוי (+ 1 2) עצמו. לכן הערך במילון שונה — הוא לא 3, אלא S-expression.

לא, קיימים ביטויים ב־L32 שלא ניתן להמיר לביטויים שקולים ב־L3 לפי שיטת 2.5. הסיבה היא שההמרה נעשית ברמה תחבירית בלבד, ללא הרצת הקוד. לכן אם מילון ב־L32 מכיל מפתח שהוא משתנה (כמו (dict (y 2)) כאשר y מוגדר כ־'b'), לא ניתן לדעת שערכו של y הוא 'b' בלי להריץ את התוכנית. כתוצאה מכך, ההמרה תניב מילון עם מפתח 'y' במקום 'b', והמשמעות של התוכנית תשתנה.

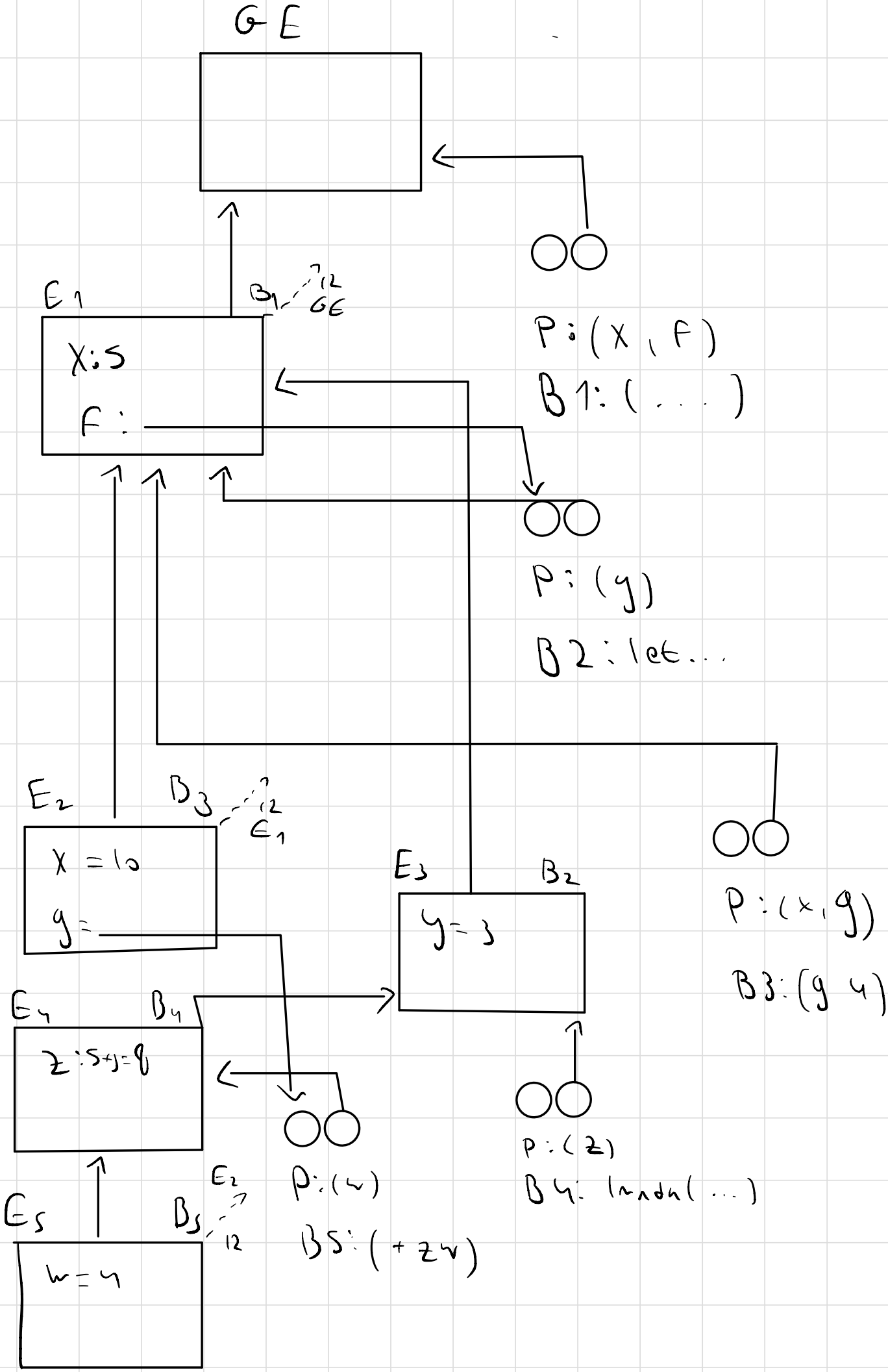
e. עבור 2.1 היתרונות זה המימוש הקל באינטרפטר מכיוון שמשתמש בפרימיטיבים פשוטים אך חסרון התחביר מסורבל חייבים quote ואין תמיכה בביטויים חיים.
עבור 2.2 היתרונות שהתחביר נוח למשתמש ותומך בביטויים חיים החסרונות זה הקושי במימוש, דורש שינוי בפארסר ובאינטרפטר, פחות גמיש לתחזוקה
עבור 2.3 היתרונות שאינו דורש שינוי בשפה החסרונות שהוא הכי איטי ומסובל טכנית.

אנחנו מעדיפים את 2.2 בעקבות היתרונות שאמרנו

<u>שאלה 4 סעיף ב</u>

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

 (
  let
   (
    (f  (h 3))
   )
   (f 2)
 )
)
```


GE
E1
B1
GE
X:5
F:
P:(X, F)
B1:(...)
P:(y)
B2:let...
E2
B3
E1
x = 10
y=
E3
B2
y=3
P:(x, y)
B3:(y 4)
E4
B4
z:5+y=8
P:(z)
B4: lnadn(...)
E5
B5
E2
12
w=4
P:(w)
B5:(+ zw)