1. ההבדל בין special form לבין primitive operator הוא באופן שבו האינטרפרטר של השפה מבצע חישוב שלהם. בעוד שהאופרטור הפרימיטיבי מחושב על ידי פונקציה מוגדרת מראש בשפת ה-meta איתה כותבים את האינטרפרטר של השפה, באמצעות הפעלת הפונקציה "applyPrimitive", לכל special form כלל חישוב כזה משלו. צורת חישוב זו גוררת הבדלים משמעותיים בין ביטויים שונים אלו. נמחיש הבדלים אלו בחישוב בהסתכלות על ביטויי הסתעפויות דוגמת if. בביטוי מדגם if, אשר בהתאם ל-Abstract Syntax של השפה, מורכב מ-3 חלקים עיקריים (test, then, else), אנו קודם כל מחשבים את חלק ה-test, ובהתאם לתוצאת החישוב מחליטים האם לחשב את then או את else, בהתאם לכלל החישוב המיוחד של ביטוי ה-if. בניגוד גמור לכך, בכל אופרטור פרימיטיבי, על מנת לבצע הפעלה שלו, יש לחשב בצורה מלאה את כל הארגומנטים שנקראו להפעלה עליו, ולא ניתן להשאיר חלקים לא מחושבים.
2. a. Eval(<Exp[] exps>, env) =>  
    let acc: Value=false value  
    for every exp in exps and acc is considered a false value  
    acc=eval(exp, env) || acc  
    return acc  
   b. Eval(<Exp[] exps>, env) =>  
    let acc: Value=false value  
    for every exp in exps  
    acc=eval(exp, env) || acc  
    return acc
3. נעדיף להשתמש בשיטה של varRef מכיוון שאז נצטרך במידת הצורך לבצע את השינוי של הוספה או הסרת אופרטור פרימיטיבי בפשטות בקוד במקום אחד (ב-applyPrimitive), ואילו בשיטה השנייה, כשנרצה לבצע פעולה שכזאת ניאלץ לבצע שינויים בשני מקומות (ב-ast וב-eval) והשינוי יהיה משמעותי יותר.
4. מעבר מ- applicative orderל-normal order יניב מס׳ יתרונות:  
   א. במקרים בהם הארגומנטים לפונ׳ הם ביטויים מורכבים שהחישוב שלהם לא תמיד נדרש, ב-normal order לא נחשב אותם בעוד שב- applicative order כן. דוגמה טובה לכך ניתן לראות בקטע הקוד הבא:

((lambda(x y z p)

y)

(( 12 34 -) ( 4 5 7 \*) 25 ( 8 9 7+)

כפי שניתן לראות, הגדרנו lambda שמקבלת ארבעה פרמטרים, כאשר שלחנו לה 3 פרמטרים שהינם ביטויים מורכבים (non special compound expressions). ביטויים אלו, כפי שקל לראות בדוגמה למעלה – לא צריכים להיות מחושבים. כתוצאה מכך, החישוב יהיה מהיר הרבה יותר באמצעות שימוש ב-normal evaluation בהשוואה ל-applicative, שכן לא נחשב את תתי ביטויים אלו כלל בעת הפעלת הלמבדה!

ב. יהיו מקרים בהם דחיית חישוב הארגומנטים תמנע זריקת חריגה כמו במקרה שהוצג בהרצאה בו אחד הארגומנטים לפונ׳ היה ביטוי מורכב שכלל חלוקה ב-0, בדוגמה, עקב השימוש ב-if לא חושב הארגומנט, מה שמנע זריקת חריגה. להלן דוגמה הפועלת על עיקרון דומה מההרצאה:

1. ((lambda (x y)
   * + 1. (if (y ? 4) 4 (+ x 4)))
       2. (/ 1 0), (+ 2 2))

כפי שניתן לראות, במקרה זה אין כל צורך בחישוב הארגומנט X אם הערך של y הוא אכן 4. במקרים רבים בהם y יהיה 4, ו-x ביטוי לא חוקי כלשהו, כמו כזה הכולל חלוקה ב-0, לא ייזרק אקספשן מחישוב הביטוי בחישוב הlambda שכן הארגומנטים של הפונקציה מחושבים רק כשהם נדרשים בnormal order, ובהתאם לתוצאת חישוב חלק ה-test יחושב לעתים חלק ה-then ולא ה-else שעשוי להוביל לשגיאה. בניגוד לכך, אם היינו פועלים בשיטת applicative, ברור שהייתה נזרקת שגיאה עוד בתחילת החישוב, בעת חישוב הארגומנטים לפני הפעלת הclosure על הפרמטרים הנשלחים.

ג. יהיו מקרים בהם דחיית חישוב הארגומנטים תמנע כניסה ללולאה אינסופית. ראינו דוגמה לכך בכיתה. אנו מצרפים בזאת דוגמה דומה אותה ננתח על מנת להמחיש זאת בבירור:

1. (define loop (lambda(x) (loop x))
2. (define return\_Five(lambda(x) 5))
3. (return\_Five(loop 26))

כפי שניתן לראות בדוגמה זאת, אם היינו פועלים בחישוב בשיטת applicative order, היינו נתקעים בloop אין סופי כחלק מחישוב הארגומנט להפעלה על הclosure של הלמבדה return\_Five, שמשום שאינה זקוקה לחישוב הארגומנט הנשלח אליה כדי להחזיר ערך, תחזיר את הערך 5 ותסיים ללא כל כשלים בעת חישוב בשיטת normal order.

1. מעבר ל- applicative orderמ-normal order יוביל לשינוי משמעותי ביעילות החישוב של תכניות בשפה שלנו (ככל הנראה יהפוך אותה לאיטית בהרבה, במרבית המקרים) נראה דוגמה מפורטת לכך בתשובתנו, כאשר בתחילה נסביר את העיקרון שעומד מאחוריה.

בשימוש ב- applicativeאנו דואגים לחישוב של כל ביטוי מורכב המהווה ארגומנט של פונ׳ בתוכניות שלנו פעם אחת בלבד, בעוד שב-normal מה שיקרה ברוב המקרים הוא חישוב של ביטויים מורכבים שוב ושוב, בשל העובדה ש״סוחבים״ את הביטוי ולא את ערך החישוב שלו עד שהוא נדרש לחישוב מכאן שבמרבית המקרים גישת applicative תהא מהירה ויעילה יותר מגישת normal order. נראה דוגמה לכך בקוד בשפת scheme:

1. (define inefficientAddition (lambda(x) (+ x x x x x x x x x x x….))
2. (inefficientAddition (\* 7 6))

בדוגמה זו, כפי שקל לראות, אם היינו פועלים בשיטת ה-normal order היינו "גוררים את הביטוי" המורכב המתקבל כארגומנט ומציבים אותו במקום כל אחד מה-xים הנ"ל. מאחר שבמקרה זה יש הפעלה של אופרטור פרמיטיבי על מספר רב של ארגומנטים, וב-normal evaluation מובן שנצטרך לחשב גם כן את כל אחד מהארגומנטים בהפעלת אופרטור פרמיטיבי על אוסף ארגומנטים. אם כן, בצורה זו – נחשב פעמים רבות את הארגומנט (ייתכן אף מאות ואלפי פעמים, הרי שפעולת ההוספה עובדת על מספר arbitrary של ארגומנטים בקלט) בניגוד לפעם אחת בלבד בחישוב בשיטת applicative.

1. נעדיף את שיטת הסביבות מכיוון שפעולת החלפת שמות המשתנים זו פעולה מסובכת ויקרה. אם נסתכל על כך לעומק: הפעולות הנדרשות כחלק מהעבודה בשיטת ההחלפה (חישוב הארגומנטים, שינוי שמות המשתנים החופשיים בגוף, ההחלפה וכו') יגררו למעשה העתקה של כל ה-AST של ביטויי למבדה, מה שיוביל לשימוש רב בזיכרון, ואף יאט את זמן הריצה של התוכנית כולה. ראינו דוגמה מפורטת להבדלים הנ"ל בחישוב בין השיטות בהרצאות, אך בשל אורכה, נקטנו בגישה שהוצגה בפרק המתאים ב-class material של הסבר מילולי להבדל זה בלבד.
2. נשתמש בדוגמה להמחשה שהוצגה לנו בהרצאות, על כך שכוללת חלוקה ב0:

(define example

  (lambda (x y)

    (if (= x 0)

0

    y)))

(define  divbyzero

  (lambda (n)

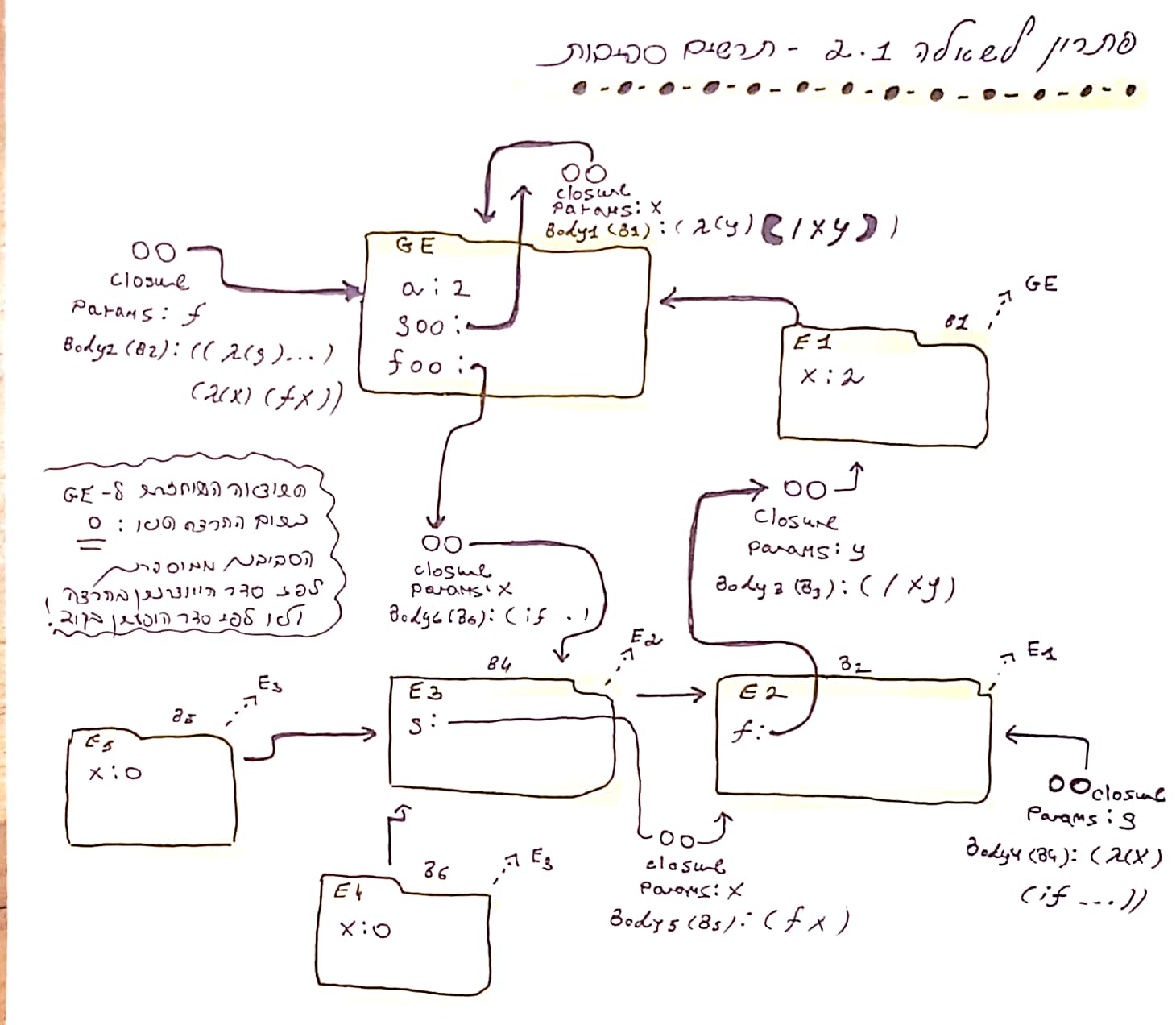
    (/ n 0))) ; division by zero!

(example 0 (divbyzero 8))

1. לא צריך פונ׳ זו מכיוון שאז הפרמטרים לא מחושבים מראש, אלא רק לפי הצורך, לכן לא צריך להפוך את הפרמטרים שכבר חושבו ל-litExp מחדש.
2. לא צריך פונ׳ זו מכיוון שכשאנו משתמשים בפונקציה "applyProc" אנו שומרים את ערכי הארגומנטים בקריאה לפונקציה בסביבה מורחבת, עם binding מתאים של שם ל-value. בצורה זו אנחנו כלל לא נדרשים לגעת בגוף הפונקציה (גוף הlambda), כך שלא נעשית החלפה ואין צורך בשינויי שמות ודומיהם. בכל פעם שנצטרך לגשת לערך של פרמטר, נחפש אותו בפשטות בסביבה. במידת הצורך, נעשה זאת בצורה רקורסיבית בסביבות המורחבות על ידי הסביבה שנפתחה.
3. ישנן שתי גישות לחישוב ביטויי let אותן למדנו בכיתה, הגישה אותה אימצנו: דוגלת בהמרה של כל ביטוי let ללמבדה השקולה לו טרם ביצוע הליך החישוב של הביטוי. במקרה זה כחלק מהליך החישוב של למבדה ייווצר closure בעת חישוב הביטוי, ולכן בכל פעם שנחשב let expression במקרה זה ניצור closure מתאים. בגישה זו דגלנו ב4L.



הגישה השנייה אותה למדנו הינה חישוב של ביטויי let באופן ישיר (ללא המרה כחלק מ-syntactic abbreviations) ובמקרה זה לא ייווצר closure לכל ביטוי let כמו בגישה הראשונה. בשיטה זו דגלנו בשפה 3L.

2.1.

2.2. (define foo lambda (goo)

(let ((a 1) (b 2))

(lambda (arg) (goo arg))))

(define goo (foo (lambda (y) (+ y 5))))

(goo 5)