

# עקרונות שפות תכנות סמסטר ב-2010

## תרגול 3: Syntax and Operational Semantics

### Topics:

1. Abstract and Concrete syntax.
2. Operational Semantics

### תחביר קונקרטי (Concrete Syntax)

התחביר הקונקרטי של שפת תכנות הוא אוסף חוקים המגדיר את השפה. התחביר הקונקרטי מגדיר את האפשרויות לשלב בין התווים והסימבולים המוכרים בשפה לכדי ביטויים בעלי מבנה חוקי בשפה. התחביר הקונקרטי של שפת scheme הוא דקדוק חסר הקשר, מצומצם ופשוט (scheme היא שפה חסרת הקשר בשונה מרוב שפות התכנות).

הערה: התחביר מתאר מהי תוכנית חוקית, אך לאמתייחס למשמעות של התוכנית או לתוצאת הרצתה. כלומר, לא כל תוכנית הנכונה מבחינה תחבירית נכונה גם סמנטית: ניתן לכתוב תוכנית ללא שגיאות תחביר (syntax) ושאינה פועלת כנדרש.

דוגמא: התחביר הקונקרטי של scheme מגדיר ביטוי IF באמצעות החוקים הבאים מתוך הדקדוק של השפה:

```
<scheme-exp> → <exp> | <define>
<exp>         → <atomic> | <composite>
<composite>  → <special> | <form>
<special>    → <lambda> | <quote> | <cond> | <if>
<if>         → '( 'if' <exp><exp><exp> )'
```

### תחביר אבסטרקטי (Abstract Syntax)

התחביר האבסטרקטי מתייחס לחלקים המרכיבים ביטויים בשפה ושיש להם משמעות בתוכנית הכתובה בשפה. התחביר האבסטרקטי מתעלם מן המאפיינים של השפה שאינם רלוונטיים למשמעות התוכנית: סדר הארגומנטים בפעולת if למשל, או הסוג המסוים של סוגרים המשמשים בכתיבת תוכנית בשפה.

דוגמא: התחביר האבסטרקטי של scheme מגדיר ביטוי IF באופן הבא:

```
<scheme-exp>:
  Kinds: <exp>, <define>
  components:
<exp>:
  Kinds: <atomic>, <composite>
  components:
<if>:
  Kinds:
  components: Predicate: <exp>
               Consequence: <exp>
               Alternative: <exp>
```

## שאלה 1

כיצד ישתנו התחביר הקונקרטי והאבסטרקטי אם נרצה ששפת scheme תתמוך בביטוי **if** מהצורה **(if x then y else z)** ?

**תשובה:**

התחביר הקונקרטי של השפה ישתנה: כדי לתמוך בצורה החדשה לבטא ביטוי **if**, נדרש להרחיב את חוק הגזירה עבור ביטוי **if** באופן הבא:

$\langle \text{if} \rangle \rightarrow \text{'(' 'if' } \langle \text{exp} \rangle \langle \text{exp} \rangle \langle \text{exp} \rangle \text{' )' | '(' 'if' } \langle \text{exp} \rangle \text{'then' } \langle \text{exp} \rangle \text{'else' } \langle \text{exp} \rangle \text{' )'}$

לעומת זאת, התחביר האבסטרקטי לא ישתנה כלל: הביטוי בצורתו החדשה מכיל את אותם חלקים כמו הביטוי בצורתו המקורית.

## שאלה 2

התחביר הקונקרטי של Scheme תומך בהגדרת משתנה לפי החוק הבא:  $\langle \text{define} \rangle \rightarrow (\text{define } \langle \text{name} \rangle \langle \text{exp} \rangle)$ . כיצד ישתנה התחביר אם נרצה להוסיף אפשרות להגדרת משתנה באופן המוצג בדוגמא הבאה: הביטוי  $(x := 3)$  יגדיר את ערך המשתנה  $x$  להיות 3.

## שאלה 3

כיצד ישתנו התחביר הקונקרטי והאבסטרקטי אם נרצה ששפת scheme תתמוך בהגדרת משתנה מטיפוס פרוצדורה בצורה הבאה:

$\langle \text{define} \rangle (\langle \text{procedure-name} \rangle \langle \text{formal-parameters} \rangle) \langle \text{body} \rangle$

כך שאת הפרוצדורה `proc1` נוכל להגדיר למשל גם באופן הבא:

```
(defineproc1 (lambda (x)
  (display x)
  (newline)
  x))
```

**תשובה:**

לתחביר הקונקרטי נוסיף את החוק הבא:

$\langle \text{define} \rangle \rightarrow \text{'(' 'define' } \langle \text{variable} \rangle \langle \text{exp} \rangle \text{' )' | '(' 'define' '(' } \langle \text{variable} \rangle \langle \text{par-sequence} \rangle \text{' )' } \langle \text{exp-sequence} \rangle \text{' )'}$

לתחביר האבסטרקטי נוסיף קטגוריה:

$\langle \text{scheme-exp} \rangle$ :

Kinds:  $\langle \text{exp} \rangle$ ,  $\langle \text{define} \rangle$ ,  $\langle \text{define-procedure} \rangle$

$\langle \text{define-procedure} \rangle$ :

**components:**    **Variable:**  $\langle \text{variable} \rangle$   
                  **Parameters:**  $\langle \text{variable} \rangle$ . Amount:  $\geq 0$ . Ordered.  
                  **Body:**  $\langle \text{exp} \rangle$ . Amount:  $\geq 1$ . Ordered.

## שיטות חישוב (Operational semantics)

### 1. תחום הגדרה (scope), מופעים חופשיים (free) וקשורים (bound). הגדרות:

- **Declaration או Binding Instance:** מופע של משתנה  $x$  הקושר את  $x$  עם ערך. בשפה `scheme`, המשתנים ברשימת הפרמטרים הפורמאליים בביטוי `lambda` הם `binding instances`. מופע של משתנה המוגדר בסביבה הגלובאלית, על ידי פעולת `define`, גם הוא `binding instance`. בדוגמא הבאה, ה- `Binding Instance` של המשתנה `even?` מופיע בשורה 1 וה- `Binding Instance` של `n` מופיע בשורה 2. **אלו הם `Declarations` של המשתנים `even?` ו-`n`.**

```
1. (define even?
2.   (lambda (n)
3.     (eq? (/ n 2) 0)))
```

- **Occurrence:** מופע של משתנה שאיננו `binding instance`. בדוג' לעיל קיים Occurrence של `n` בשורה מספר 3.
- **Scope:** תחום ההגדרה של `binding instance` (משתנה קושר). חלק הקוד של התוכנית בו Occurrences של המשתנה מתייחסים אל ה- `binding instance`. בשפה `Scheme`, ה- `scope` של פרמטר פורמאלי בביטוי `lambda` הוא גוף ה- `lambda`. ה- `scope` של משתנה בביטוי `define` הוא כל התוכנית החל ממקום הגדרתו והלאה (`Universal Scope`). בדוגמא לעיל, ה- `scope` של המשתנה `even?` הוא ה- `Universal Scope` ואילו ה- `scope` של `n` הוא גוף ה- `lambda`, כלומר שורה 2.
- **Bound Occurrence:** נאמר כי מופע של משתנה  $x$  הוא מופע קשור בביטוי נתון, אם קיים משתנה קושר ל-  $x$  בתוך הביטוי. בדוגמא לעיל, `Bound Occurrence` של מופיע בשורה 3 וקשור ע"י ה- `Binding Instance` של `n` משורה 1.
- **Free Occurrence:** נאמר כי מופע של משתנה  $x$  הוא מופע חופשי בביטוי נתון, אם לא קיים משתנה קושר ל-  $x$  בתוך הביטוי. בדוגמא לעיל, המופעים של `eq?` ושל אופרטור החילוק (`/`), הם מופעים חופשיים בתוך הביטוי.

### דוגמא (1):

נתבונן בפרוצדורה `expt?` הבודקת האם מספר נתון,  $n$ , הוא חזקה של 2. לכל משתנה קושר (`Binding instance`) ציין את תחום ההגדרה (`Scope`) שלו ואת המופעים הקשורים (`Bound instance`).

```
1. (define even? (lambda (n)
2.   (eq? (/ n 2) 0)))

3. (define expt? (lambda (n)
4.   (cond ((= n 1) #t)
5.         ((even? n) (expt? (/ n 2)))
6.         (else #f)))
```

Binding instance	Binding instance line#	Scope	Bound occurrences' line #
<code>even?</code>	1	<code>define</code> , <code>Universal Scope</code>	5
<code>n</code>	1	<code>lambda body</code> , line 2	2
<code>expt?</code>	3	<code>define</code> , <code>Universal Scope</code>	5
<code>n</code>	3	<code>lambda body</code> , lines 4-6	4,5

## דוגמה (2):

```
1. (define p1 (lambda (x y)
2.     (+ (* x y)
3.     (+ (* 2 x)
4.     (+ (* 2 y) 22))))))
5. (define p3 (lambda (x)
6.     (+ (p1 x 0)
7.     (+ (p1 y 1) (p2 x))))))
```

Binding instance	Binding instance Line #	Scope	Bound occurrences' Line #

המשתנה p2 המופיע בשורה 7 אינו מופיע ב-scope של מופע של p2 הקושר אותו לערך (binding instance) ולכן הוא מופע חופשי (free occurrence). מאותם שיקולים, המשתנה y המופיע בשורה 7 הוא free occurrence.

**נשים לב:** ההתייחסות אל מופע של משתנה כחופשי או כקשור היא **ביחס לביטוי מסוים**. אם היינו "מרחיבים" את הביטוי בדוגמא האחרונה כך שיכלול שורות קוד נוספות מן התוכנית, יתכן שכבר לא היינו יכולים לומר שהמופע של p2 בשורה 7 הוא מופע חופשי.

## 2. Renaming & Substitution

**כיצד נבצע Renaming באופן קונסיסטנטי (כך שלא תשתנה משמעות הביטוי)?** כדי לשנות שם של מופע מסוים בביטוי נתון, על המופע להיות **מופע קשור** בביטוי. יש לשנות את שם המופע הקושר (binding instance) ואת כל שמות המופעים הקשורים אליו (bound occurrences). לדוגמא:

```
(lambda (x) x) → (lambda (y) y)
```

בביטוי המסומן במסגרת לעיל, החלפנו את השם של המופע x: החלפנו את המשתנה הקושר ואת כל המופעים הקשורים אליו. **האם נוכל לשנות את שם המופע y ל-z בביטוי הבא שבמסגרת?**

```
(define y 10)
```

```
(lambda (x) (+ x y))
```

מדוע אנו נדרשים לבצע renaming? נתבונן בביטוי הבא המתאר הפעלה:

1.  $(\lambda (x)$
2.  $(\lambda (y)$
3.  $(x y))$

4.  $(\lambda ()$
5.  $(y w))$

אם נבצע הצבה מבלי לבצע renaming, נקבל את הביטוי הבא:

```
(lambda (y)
  ( (lambda () (y w) ) y))
```

הבעיה נובעת מכך שבביטוי במסגרת העליונה המופע של  $y$  חופשי ואילו בביטוי במסגרת התחתונה הוא קשור. כפי שראינו בהרצאה, כדי להעריך את הביטוי נכונה, נרצה לחלק למשתנים הקיימים שמות חדשים (שאינם מופיעים בקוד) באופן קונסיסטנטי ורק לאחר מכן לבצע את ההחלפה:

**Substitute[x,(lambda()(y w)),(lambda (y) (x y))]:**

1. Renaming:  
 $[(\lambda ()(y w))]=(\lambda ()(y w))$   
 $[(\lambda (y)(x y))]=(\lambda (y2) (x y2))$
2. Substitution:  
 $[(\lambda (y2) (x y2))]=(\lambda (y2) ((\lambda ()(y w)) y2))$

כדי להחליף מופע חופשי של משתנה אבתוך ביטוי  $e$  על ידי ביטוי  $v$ , נסמן את ההחלפה:  $sub(x,v,e)$  ונפעל באופן הבא:

1. נבצע Renaming באופן קונסיסטנטי עבור הביטויים  $v$  ו- $e$ .

2. נחליף את כל המופעים החופשיים של המשתנה  $x$  בביטוי  $e$  (החדש) על ידי הביטוי  $v$  (החדש).

הערה: תמיד נבצע Renaming באמצעות שימוש ב- $subscript$ . לדוגמא, אם נרצה את שם המופע של  $x$  נשתמש בשמות  $x1, x2, \dots$

### Evaluation Order .3

הפונקציה  $eval[e]$  מקבלת ארגומנט (scheme-expression), מעריכה (evaluates) אותו ומחזירה ערך שהוא Scheme Type (עד כה הכרנו את - Scheme Types: Number, Boolean, Symbol, Procedure). ישנם שני אלגוריתמים המממשים את  $eval(e)$ :

1. **Applicative-eval(e)**: האלגוריתם נוקט בשיטת eval-sub-reduce, משמע חישוב ערכי ביטויי האופרנדים (eval) נעשה בטרם

שלב ההחלפה (sub). כאשר נעריך ביטוי מורכב (composite), תחילה יחושבו ערכי כל הפרמטרים ולאחר מכן תבוצע ההצבה של

ערכי הפרמטרים. אלג' זה הוא המנגנון הסטנדרטי בשפות **scheme, pascal, c**.

2. **Normal-eval(e)**: האלגוריתם נוקט בשיטת sub-reduce, בה ההצבה מתבצעת עם ביטויי הפרמטרים עצמם **ולא עם הערכים**

המחושבים שלהם. כאשר נעריך ביטוי מורכב (composite), הפרמטרים של הביטוי לא יוחלפו בערכים המחושבים שלהם עד אשר

לא תהיה ברירה אלא להחליפם. שיטת הערכה זו מכונה גם הערכה עצלה (Lazy approach for evaluation).

מתי אין ברירה? בהערכה לפי normal order, הפרמטרים של הביטוי יוחלפו בערכים המחושבים שלהם במקרים הבאים:

א. כאשר יש צורך בהחלטה חישובית לגבי הסתעפות, למשל הערכה של תנאי ב-**cond**.

ב. כאשר מעריכים את הביטוי הראשון בביטוי מורכב (composite).

ג. כאשר נדרש ערך הביטוי לשם חישוב הערך המוחזר מהפעלת פונקציה פרימיטיבית (+, \*, /, number?, ...).

**דוגמא (3):** פעולת Sub לפי Normal order (הצבעים מבהירים את סדר הפעולות!).

```
( (lambda (z)
  (+ ( (lambda (x z)
        (+ x z 2))
      z 0)
    1) )
(lambda (z) z) 3))
```

**sub[z, ((lambda (z) z) 3)], (+ ((lambda (x z) (+ x z 2)) z 0) 1)]**

**1. Renaming:**

a.  $[(\text{lambda } (z) z) 3] = (\text{lambda } (z1) z1) 3$

b.

$$\left[ \begin{array}{c} +((\text{lambda } (x z) \\ (+ x z 2)) z 0) \\ 1) \end{array} \right] = \left[ \begin{array}{c} +((\text{lambda } (x1 z2) \\ (+ x1 z2 2)) z 0) \\ 1) \end{array} \right]$$

**2. Substitution:**

$$\left[ \begin{array}{c} +((\text{lambda } (x1 z2) \\ (+ x1 z2 2)) z 0) \\ 1) \end{array} \right] = \left[ \begin{array}{c} +((\text{lambda } (x1 z2) \\ (+ x1 z2 2)) \\ ((\text{lambda } (z) z) \\ 3) \\ 0) \\ 1) \end{array} \right]$$

נבחין כי במקרה זה ניתן לוותר על שלב 1.a של חלוקת השמות החדשים. במקרים מסוג זה נוכל לדלג על שלב ה-Renaming ולבצע את ההחלפה. הסיבה ששלב ה-Renaming אינו הכרחי בדוגמא זו היא כיוון שבביטוי שהצבנו במקום z אין מופעים חופשיים של משתנים שעלולים להיקשר על ידי פרמטר בעל שם זהה בפונקציה עוטפת. בדוגמא הבאה נראה כי ה-Renaming הכרחי, ואם לא נבצעו נקבל ערך מוחזר שגוי.

**דוגמא (4):** הדוגמא תציג תהליך substitution לפי Normal Order ותמחיש את החיוניות של שלב ה-Renaming. נבצע Substitution לפי Normal Order, ללא יישום שלב ה-Renaming, בביטוי הבא המתאר הפעלה:

```
((lambda (f)
  (lambda (x) (sqrt (f x)))))

(lambda (y) (+ x y))
)

sub[f, (lambda (y) (+ x y)), (lambda (x) (sqrt (f x)))]

==>(lambda (x)
  (sqrt ((lambda (y) (+ x y))x)))

==>(lambda (x) (sqrt (+ x x)))
```

אם נפעיל את הביטוי שקיבלנו על הקלט 4.5, נקבל את התוצאה הבאה:

```
>((lambda (x) (sqrt (+ x x))) 4.5)
3.0
```

התוצאה שהתקבלה היא כמובן שגויה, כיוון שבביטוי המקורי:

```
((lambda (f)
  (lambda (x) (sqrt (f x))))

(lambda (y) (+ x y))
)
```

מופעי x המודגשים הם מופעים הקשורים למשתנים שונים בעלי שם זהה: הראשון הוא מופע של הפרמטר x של הפונקציה (המסומנת בקו תחת) ואילו השני הוא מופע חופשי. כאשר נבצע renaming בצורה נכונה לביטוי נקבל את הביטוי הבא:

```
(lambda (x1) (sqrt (+ x x1)))
```

אם נפעיל אותו על הקלט 4.5, נקבל שגיאה כיוון שאין ערך למשתנה x:

```
>((lambda (x1) (sqrt (+ x x1))) 4.5)
Undefined variable x...
```

## :Applicative-eval האלגוריתם

- :: 1. Signature: applicative-eval(e)
- :: 2. Purpose: to evaluate the expression e by Applicative order evaluation
- :: 3. Type: <Scheme-EXP> --> Scheme-TYPE
- :: 4. Example: (applicative-eval (if (and (> 2 3) (not #t)) 1 2)) should produce 2
- :: 7. Tests: (if (and (> 2 3) (not #t)) 1 2) ==>2

### applicative-eval[e] =

- I. if e is **atomic** expression:
  1. if e is a **number** or a **boolean**: e evaluates to itself.
  2. if e is a **symbol**:
    - a. if e has a binding in the global environment, its value is the binding value.
    - b. otherwise, e must be a variable denoting a primitive procedure or a special operator. It evaluates to its built-in code.
- II. Otherwise, e is a **composite** expression: e = ( e0 e1 ... en ) where ( n >= 0 ):
  1. If e0 is a **special operator**:  
applicative-eval[e] is defined by the special evaluation rules of e0 (see below).
  2. otherwise:
    - a. **evaluate**: compute applicative-eval[ei] = ei' for all ei.
    - b. if e0' is a primitive procedure:  
eval[e] = system application e0'(e1', ..., en').
    - c. otherwise, e0' is a **closure (user procedure)** constructed by (lambda (x1 ... xn) b1 ... bm)
      - i. **substitute**: compute sub[xi, ei', bj] = bj' for 1 <= i <= n, 1 <= j <= m  
;; The substitution is by no specific order and preceded by renaming.
      - ii. **reduce**: applicative-eval[b1'], ..., eval[bm'].
      - iii. **return**: applicative-eval[e] = eval[bm'].

### SPECIAL OPERATORS EVALUATION RULES:

1. e = (define x e1)  
applicative-eval[e1] = e1', add to the global environment the binding <x, e1'>
2. e = (lambda (x1 x2 ... xn) b1 ... bm) at least one bi is required.  
Construct a closure with **parameter sequence**: x1, ..., xn (can be empty) and **body**: b1, ..., bm.
3. e = (quote e1)  
applicative-eval[e] ==> e1
4. e = (cond (p1 e11 ...) ... (else en1 ...)):  
if applicative-eval[p1] != #f:  
    applicative-eval[e11], ... , applicative-eval[en1].  
    applicative-eval[e] = eval[en1].  
otherwise, continue with p2 in the same way.  
if for all pi-s applicative-eval[pi] = #f:  
    applicative-eval[en1], ... applicative-eval[enm].  
    applicative-eval[e] = applicative-eval[enm].
5. e = (if p con alt):  
if applicative-eval[p] != #f  
    then eval[e] = eval[con]  
    else eval[e] = eval[alt]



## דוגמא (5):Applicative order reevaluation:

```

applicative-eval[ ( (lambda (x y) (* (+ x y) y)) 1 (+ 1 2) ) ]
  applicative-eval [ (lambda (x y) (* (+ x y) y)) ] ==> #<closure (x y) (* (+ x y) y)>
  applicative-eval [1] ==> 1
  applicative-eval [(+ 1 2)]
    applicative-eval [+]==> #<primitive-procedure +>
    applicative-eval [1] ==> 1
    applicative-eval [2] ==> 2
  ==> 3
sub(x, 1, (* (+ x y) y)) ==> (* (+ 1 y) y)
sub(y, 3, (* (+ 1 y) y)) ==> (* (+ 1 3) 3)

```

הערה: דילגנו על שלב ה renaming כיוון שכל המשתנים בביטוי המוערך קשורים, ואין משתנים כארגומנטים.

reduce:

```

applicative-eval [( * (+ 1 3) 3)]
  applicative-eval [*]==> #<primitive-procedure *>
  applicative-eval [(+ 1 3)]
    applicative-eval [+]==> #<primitive-procedure +>
    applicative-eval [1] ==> 1
    applicative-eval [3] ==> 3
  ==> 4
  applicative-eval [3]==> 3
==> 12

```

## דוגמא (6): לא את כל הביטויים ניתן להביא לצורה פשוטה.

```

((lambda (x) (x x)) (lambda (x) (x x)))
sub[x, (lambda (x) (x x)), (x x)]
1. Renaming [(lambda (x) (x x))] = (lambda (x1) (x1 x1))
2. substitution[(x x)] = ((lambda (x1) (x1 x1)) (lambda (x1) (x1 x1)))

```

הביטוי המתקבל לאחר ההצבה זהה לביטוי המקורי! תהליך ה-evaluation לפי האלגוריתם applicative-eval יימשך לעד:

```

applicative-eval[ ( (lambda (x) (x x)) (lambda (x) (x x)) ) ]
  applicative-eval [ (lambda (x) (x x)) ] ==>#<closure(x)(x x)>
  applicative-eval [ (lambda (x) (x x)) ] ==>#<closure(x)(x x)>
sub(x, #<closure(x)(x x)>, (x x)) ==> (#<closure(x)(x x)>#<closure(x)(x x)>)

```

## דוגמא (7): נתבונן בהפעלה של הפרוצדורה (lambda (x y) y) עלהאופרנדים ((lambda(x)(x x)) (lambda(x)(x x))) -1:

```

( (lambda (x y) y) ( (lambda (x) (x x)) (lambda (x) (x x)) ) 1) = (e0 e1 e2)
applicative-eval[ ( (lambda (x y) y) ( (lambda (x) (x x)) (lambda (x) (x x)) ) 1) ]
  applicative-eval[ ( (lambda (x y) y) ) ] ==>#<closure(x y)(y)>
  applicative-eval[ ( (lambda (x) (x x)) (lambda (x) (x x)) ) ] ==>endlessevaluation...

```

בהערכת הביטוי לפי applicative-eval, האלגוריתם יעריך תחילה רקורסיבית את e1, e0, e2 ולאחר מכן יוחלפו המופעים החופשיים של x ושקל-y בגוף הפרוצדורה של e1-e2. כאשר האלג' יגיע להעריך את e1, החישוב יימשך לעד כפי שראינו בדוגמא (6) ולא נוכל להעריך את הביטוי. נשים לב שאין באמת צורך לחשב את e1, כיוון שהתוצאה הסופית אינה תלויה בערכו, כלומר לו החלפנו את המופעים החופשיים של x ושקל-y בגוף הפרוצדורה על ידי e1-e2 עצמם ולא על ידי ערכיהם, החישוב היה מסתיים ומחזורי. דרך פעולה זו היא המשמשת את האלג' normal-eval, בהערכת הביטוי לפי אלג' זה, חישוב הערך של e1 נחסך ומתקבלת התוצאה 1.

**האלגוריתם Normal-eval:** פעולת האלגוריתם זהה לזו של applicative-eval פרט לעובדה שהוא נמנע מלהעריך ביטויים עד אשר אין ברירה. גישה זו להערכה של ביטויים מכונה גם lazy evaluation. ההבדל העיקרי בין האלגוריתמים הוא בהסרת שורה II.2.a. שורות הקוד הבאות מציגות רק את השינוי באלגוריתם שראינו קודם לקבלת האלגוריתם normal-eval:

**II. Otherwise, e is a composite expression:  $e = (e_0 e_1 \dots e_n)$  where  $(n \geq 0)$ :**

1. if  $e_0$  is a special operator:

normal-eval[e] is defined by the special evaluation rules of  $e_0$ .

2. **evaluate:** compute normal-eval[e<sub>0</sub>] = e<sub>0</sub>'.

בביטוי מורכב, האלגוריתם מעריך תחילה רק את הפרמטר הראשון, e<sub>0</sub>, ולא את כל הפרמטרים כבאלגוריתם applicative-eval.

b. if  $e_0'$  is a primitive procedure:

evaluate: compute normal-eval[e<sub>i</sub>] = e<sub>i</sub>' for all e<sub>i</sub>.

normal-eval[e] = system application e<sub>0</sub>'(e<sub>1</sub>', ..., e<sub>n</sub>').

c. otherwise,  $e_0'$  is a closure (user procedure) constructed by (lambda (x<sub>1</sub> ... x<sub>n</sub>) b<sub>1</sub> ... b<sub>m</sub>)

i. **substitute (expansion):** compute sub[x<sub>i</sub>, e<sub>i</sub>, b<sub>j</sub>] = b<sub>j</sub>' for 1 ≤ i ≤ n, 1 ≤ j ≤ m

; The substitution is by no specific order and preceded by renaming.

ii. **reduce:** applicative-eval[b<sub>1</sub>'], ..., eval[b<sub>m</sub>'].

iii. **return:** applicative-eval[e] = eval[b<sub>m</sub>'].

ההחלפה מתבצעת עם הביטויים עצמם ולא עם הערכים המחושבים שלהם (e<sub>i</sub>' במקום e<sub>i</sub>).

**דוגמא (8): Normal order evaluation**

```
(define test (lambda (x y) (if (= x 0) 0 y)))
(define zero-div (lambda (n) (/ n 0))) ; division by zero!

normal-eval [(test 0 (zero-div 5))]
  normal-eval [test] ==> <closure of (lambda (x y) (if (= x 0) 0 y))>
sub [x, 0, (if (= x 0) 0 y)] ==> (if (= 0 0) 0 y)
sub [y, (zero-div 5), (if (= 0 0) 0 y)] ==> (if (= 0 0) 0 (zero-div 5))
reduce:
normal-eval [(if (= 0 0) 0 (zero-div 5))]
  normal-eval [= 0 0]
    normal-eval [=] ==> #<primitive-procedure =>
    normal-eval [0] ==> 0
    normal-eval [0] ==> 0
  ==> #t
normal-eval [0] ==> 0

==> 0
```

נשים לב כי הביטוי (zero-div 5) לא הוערך במהלך הריצה! לוי הערכנו ביטוי זה (למשל אם היינו מחשבים את ערך הביטוי לפי applicative-eval) היינו מקבלים את השגיאה "division by zero" מה- interpreter. לעומת זאת, כאן קיבלנו ערך מוחזר – 0. באופן דומה, אם נחזור לדוגמא (7) ונבצע הערכה לפי normal-eval, נמנע מלולאה אינסופית ונקבל את הערך המוחזר 1.

## Side Effects & Summary

פונקציה המייצרת Side Effect היא פונק' אשר הערכתה **משנה מצב** (ללא קשר לערך ההחזרה שלה). למשל, פונק' המשנה ערך של משתנה גלובאלי, או של אחד הארגומנטים שלה, פונק' המדפיסה למסך, וכדומה.

כדי לדעת לפי איזה מן האלגוריתמים applicative-eval או normal-eval פועל ה-`interpreter` אנו משתמשים, נוכל להשתמש בפרוצדורה המייצרת Side Effect. לפרוצדורה הפרימיטיבית `display` אין ערך מוחזר, אך היא מבצעת הדפסה למסך, כלומר מייצרת Side Effect.

```
(define (lambda (x) 1))
>(f (display 2))
?
```

מה יהיה הפלט המתקבל מן ההפעלה `(f (display 2))` בהנחה כי ה-`interpreter` פועל לפי `Applicative-order`?

מה יהיה הפלט המתקבל מן ההפעלה `(f (display 2))` בהנחה כי ה-`interpreter` פועל לפי `Normal-order`?

### לסיכום `Applicative evaluation` ו-`Normal evaluation`:

- תחום ההגדרה של `normal-eval` מכיל את תחום ההגרה של `applicative-eval` ולכן על תחום הגדרה זה האלגוריתמים שקולים (כפונקציות, לא כפרוצדורות)! במילים אחרות, אם שני האלגוריתמים מסיימים את פעולתם בזמן סופי, אזי הם מחזירים אותו ערך.
- אם האלגוריתם `applicative-eval` מסיים, אזי גם האלגוריתם `normal-eval` מסיים.
- `applicative-eval` עלול לחשב ערכים אשר אין צורך אמיתי בחישובם לשם הערכת ביטוי הקלט.
- `normal-eval` עלול לחזור על אותו חישוב מספר רב של פעמים: כיוון שמחשבים את ערכו של ביטוי רק "כשאינ ברירה", יתכן כי עד שהגענו לשלב בו יש לחשב אותו, הוא פעפעע (`propagated`) אל מקומות נוספים (ואולי רבים) בקוד. כל אחד מאלו יעבור הערכה בפני עצמו למרות שהביטויים שפעפעו נבעו מאותו ביטוי יחיד. (ראה דוגמא בהרצאה: הפעלת `((sum-of-squares 5))`.)
- קיימם ביטויים עבורם `normal-eval` יחזיר ערך, אך `applicative-eval` יכנס ללולאה אין סופית. לדוגמא, בביטוי הבא, עבור `normal-eval` נקבל 5 ועבור `applicative-eval` נתקע בלולאה אין סופית:

```
(define f (lambda (x) (f x)))
(define g (lambda (x) 5))
>(g (f 0))
```

- ניתן לזהות את סוג האלגוריתם בו פועל ה-`evaluator` ע"י שימוש ב-`side-effects`.

## High Order Procedures revisited

הפרוצדורה `exp-iter` מחשבת תוך תהליך איטרטיבי את פעולת החזקה:

```
(define exp-iter
  (lambda (b e acc)
    (cond ((= e 0) acc)
          (else (exp-iter b (- e 1) (* b acc))))))
```

```
(define exp
  (lambda (b e)
    (exp-iter b e 1)))
```

כיוון שיש להעביר את הערך 1 (האיבר הניטרלי לכפל) כפרמטר עבור `exp-iter`, הוספנו את הפרוצדורה `exp` המקבלת רק את פרמטרי הקלט ה-"אמיתיים" (הבסיס והחזקה) ומצרפת אליהם את הפרמטר 1 כאשר היא קוראת לפרוצדורה `exp-iter`. הבעיה היא שצורה זו לא מונעת שימוש בפרוצדורה `exp-iter` המוגדרת גלובלית. איך ניפטר מפרוצדורת העזר בסביבה הגלובלית?

**ניסיון ראשון:** ננסה להגדיר את `exp-iter` כמשתנה מקומי.

```
(define exp
  (lambda (b e)
    (let ((exp-iter (lambda (b e acc)
                     (cond ((= e 0) acc)
                           (else (exp-iter b (- e 1) (* b acc))))))
      (exp-iter b e 1))))
```

מה הבעיה? מהו ה `scope` של המשתנה המקומי לפי אופן הישוב של `let` (ניזכר כי `let` הוא syntactic sugar להפעלה של פרוצדורה).

```
(define exp
  (lambda (b e)
    ((lambda (exp-iter)
      (exp-iter b e 1))
     (lambda (b e acc)
       (cond ((= e 0) acc)
             (else (exp-iter b (- e 1) (* b acc))))))))
```

**פתרון:** נעביר את הפרוצדורה `exp-iter` לעצמה כפרמטר. **הסבר:** המשתנה `iter` הוא פרוצדורה המקבלת את הבסיס, החזקה, צובר ופרוצדורה נוספת אשר אותה היא מפעילה! כך אנו פותרים את הבעיה, אנו מפעילים את הפרוצדורה ומעבירים לה את עצמה.

```
(define exp
  (lambda (b e)
    (let ((iter (lambda (b e acc iter)
                 (cond ((= e 0) acc)
                       (else (iter b (- e 1) (* b acc) iter))))))
      (iter b e 1 iter))))
```

נשים לב כי אפשר להוריד גם את הפרמטר `b`, משום שהוא נמצא בתחום ההגדרה של הפרוצדורה הפנימית.

```
(define exp
  (lambda (b e)
    (let ((iter (lambda (e acc iter)
                 (cond ((= e 0) acc)
                       (else (iter (- e 1) (* b acc) iter))))))
      (iter e 1 iter))))
```