שפות תכנות

תרגול 1

הורדת תוכנת הקורס Racket והתקנת שפת הקורס PL

- הורדת DrRacket מהאתר - https://download.racket-lang.org/racket-v7.7.html.
  ב-Distribution יש לבחור - Racket
  ב-Platform יש לבחור בהתאם למערכת ההפעלה
  ב-Variant יש לבחור - Regular
- התקנת PL:
  - פותחים את DrRacket.
  - בתפריט יש לבחור - File, ולאחר מכן יש לבחור - Install .plt File...
  - בחלון שנפתח בתגית ה-URL מזינים את הכתובת הבאה:
    http://pl.barzilay.org/pl.plt

הורדת תוכנת הקורס Racket והתקנת שפת הקורס PL

- הגדרה אוטומטית של השפה PL וכיסוי בדיקות לקוד:
  - בתפריט יש לבחור - Language, ולאחר מכן יש לבחור - Choose Language…
  - בחלון שנפתח יש לבחור - The Racket Language, וללחוץ על Show Details
  - ב-Dynamic Properties יש לסמן - Syntactic test suite coverage
    * אפשרות זו נועדה לוודא שהבדיקות מכסות את כל הקוד
  - ב-Automatic #lang line יש להזין - #lang pl
    * בכל קובץ תוכנית של Racket יש לציין את השפה בה נכתב הקוד, אפשרות זו גורמת לכך שכל קובץ חדש שיפתח בתחילת הקוד תתווסף הצהרת השפה (PL) אוטומטית
- לאחר אישור ההגדרות כדאי להריץ דוגמא לבדיקת תקינות ההגדרות

```
#lang pl
```

Definitions Area

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: pl; memory limit: 128 MB.
>
```

Interaction Area

- Definitions Area: גוף התוכנית – הגדרות, פונקציות וכו'. את קובץ ההגדרות הזה אפשר לשמור ולהשתמש בהגדרות הללו שוב ושוב. סיומת קובץ כזה היא "plt.".
- Interaction Area: קלט ופלט.

כתיבה ב-Racket

<u>עצמים בסיסיים:</u>

- Booleans: true, false
- Numbers: 1, 0.5, ½, 1+2i
- Strings: ”apple” (פתיחת משפט וסגירתו בעזרת מרכאות כפולות)
- Symbols: 'apple (גרש בודד ללא רווחים)
- Characters: #\a, #\b

כתיבה ב Racket

<u>כתיבת ביטויים ואופרטורים:</u>

- ב-Racket האופרטורים הם Prefix ועטופים בסוגריים לדוגמא:
  - חיבור: (+ 1 2)
  - חיסור: (- 1 2)
  - שרשור מחרוזות: (string-append "a" "b")
- באופן כללי ביטויים ב Racket מתחלקים ל 3:
  - **עצמים**: נכתבים ללא סוגריים (כגון מספרים מחרוזות וכו')
  - **ביטויים הכוללים אופרטורים**: (<expression1> <expression2>...)
    <expression1> - הפונקציה (לדוגמא: +, string-append, *, < וכו')
    ניתן לקבל מידע על אופרטור בעזרת קליק ימני, "...“ Search in Help Desk for
  - **ביטויים מיוחדים המשתמשים במילים שמורות** כגון: התניות – (cond <exp>...), (if <exp>...), השמה – (define <name> <exp>), (let <name> <exp>), הכרזה – (: <name> : ...), ועוד

➢ **הערה: ב Racket הסימנים +,-,* וכו' נחשבים כפונקציות לכל דבר**

כתיבה ב Racket

<u>כתיבת הערות:</u>

- כתיבת שורה בודדת תעשה בעזרת ;; לדוגמא:

```
;; comment
```

- כתיבת הערה בהמשך לשורת קוד תעשה בעזרת ; לדוגמא:

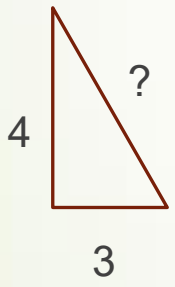
```
(+ 5 6)     ; 5 + 6 = 11
```

- כתיבת בלוק הערות יעשה בעזרת |# ו- #| לדוגמא:

```
#|
        This program does…
|#
```

דוגמא

```
#lang pl
#|
    Assume that the two short sides of a right triangle
    have length 3 and 4.
    What is the length of the long side?
```




```
    Solution using Pythagoras theorem.
|#
(sqrt (+ (* 3 3) (* 4 4)))          ; The answer is: 5
```

כיצד Racket מפרש ביטויים

- פירוש הביטוי (+ 2 (* 3 4) (- (+ 1 2) 3))

Expression

Operator

Operands

כיצד Racket מפרש ביטויים

- (+ 2 (* 3 4) (- (+ 1 2) 3))
- (+ 2 12 (- (+ 1 2) 3))
- (+ 2 12 (- 3 3))
- (+ 2 12 0)
- 14

התניות

- (if <condition-expr>
  <positive-expr>
  <negative-expr>)
- (if (< 2 3) 10 20)
- 10

Question (must produce Boolean)

True answer

False answer

התניות

- (cond
  [< condition > < to do expression >]
  …
  [< condition > < to do expression >]
  [else < else expression >])
- Racket מאפשר לכתוב cond ללא ברירת מחדל (else)
  אבל זו לא הדרך נכונה, ויש תמיד להוסיף את ברירת המחדל else לכיסוי של כל מקרה.
- לדוגמא:
- (cond
  [(eq? 'a 'b) 0]
  [(eq? 'a 'c) 1]
  [else 2])
- 2

התניות

- (cond
  [(and #t #f) 1]
  [(or #t #f) 2]
  [else 3])
- 2
- (cond
  [(not (and (not #t) (not (and #t #f)))) 1]
  [#t 2]
  [else 3])
- 1
- #|Optimization
- $\underbrace{(not\underbrace{(and\ \underbrace{(not\ \#t)}_{\#f}(not\ (and\ \#t\ \#f)))}_{\#f})}_{\#t}$

  |#

הגדרת קבועים ופונקציות

<u>הגדרת קבוע:</u>

- (define <name> <expression>)
  - <name> - שם הקבוע
  - <expression> - הערך (לאחר הערכה)
- (define PI 3.14)
- <u>הגדרת פונקציה:</u>
- (define (<function name> <arg1>…) <expression>)
  - (<function name> <arg1>…) - שם הפונקציה והארגומנטים
  - <expression> - גוף הפונקציה
- (define (Not a)
  (cond
  [a #f]
  [else #t]))

הגדרת הפונקציה - הכרזה

- כאשר מגדירים פונקציה יש להגדיר מה סוג הקלט ומה סוג הפלט (איזה סוג של משתנים הפונקציה יכולה לקבל ואיזה סוג של משתנים הפונקציה מחזירה)
- לדוגמא:

- $f(x) = x(x+1)$
- (: f : Number -> Number)
- (define (f x)
  (* x (+ x 1)))
- (f 3) ;12

דוגמא 2

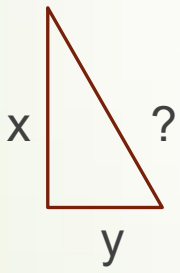
- $f(x) = \begin{cases} 0 \le x \le 5 & 1 \\ else & 0 \end{cases}$
- (: f : Number -> Number)
- (define (f x)
  (if (and (>= x 0) (<= x 5)) 1 0))
- (test (f 0) => 1)
- (test (f 5) => 1)
- (test (f 3) => 1)
- (test (f 5.1) => 0)
- (test (f -0.1) => 0)

דוגמא 3

- $f(x) = \begin{cases} 0 \le x < 5 & 2 \\ 5 \le x < 10 & 4 \\ 10 \le x & 7 \end{cases}$
- (: f : Number -> Number)
- (define (f x)
  (cond
    [(and (>= x 0) (< x 5)) 2]
    [(and (>= x 5) (< x 10)) 4]
    [(>= x 10) 7]
    [else (error 'f "The function is not defined in ~s" x)]))
- (test (f 0) => 2)
- (test (f 5) => 4)
- (test (f 10) => 7)
- (test (f -1) =error> "The function is not defined in -1")

הגדרת פונקציה - דוגמא

```
#lang pl
#|
    Build a function that takes as arguments the two short sides of a right triangle.
    The function outputs the length of the long side.
```




```
|#
```

פתרון

- הסבר כללי על הפונקציה ופתרון מתמטי
- #|
  Function ‘pyta’ takes as arguments (Naturals) x and y, the short sides of a right triangle.
  The function return the length of the long side (Number)
  Ans = sqrt(x^2 + y^2)
  |#
- הכרזה על הפונקציה, סוג הקלט וסוג הפלט
- (: pyta : Natural Natural -> Number)
- הגדרת הפונקציה
- (define (pyta x y) (sqrt (+ (* x x) (* y y))))
- ביצוע בדיקות!
- (test (pyta 3 4) => 5) ; sqrt(3^2 + 4^2) = 5
  (test (pyta 5 12) => 13) ; sqrt(5^2 + 12^2) = 13
  (test (pyta 8 15) => 17) ; sqrt(8^2 + 15^2) = 17

שפות תכנות

תרגול 2

רשימות

- רשימה ריקה: null או ()‘
- לבדוק האם הגענו לסוף הרשימה (רשימה ריקה): null?
- רשימה: או רשימה ריקה (null) או הגדרה רקורסיבית של צמד (cons) אשר האיבר השני בו הוא רשימה או איחוד איברים בעזרת הפונקציה list.
- דוגמאות:

  - (cons 1 (cons 2 null))
  - (cons 1 (cons 2 '()))
  - (list 1 2)
  - ‘(1 2)

  - (cons 2 1) ; Not valid in PL

רשימות

- לבדיקה האם אובייקט מסוים הוא מסוג רשימה ניתן להשתמש בפונקציה list?.
- דוגמאות:

- > (list? '(1 2))
- #t
- > (list? (cons 1 (cons 2 '())))
- #t
- > (list? (cons 1 null))
- #t

זוג

- לבדיקה האם אובייקט מסוים הוא צמד ניתן להשתמש בפונקציה pair?.
- דוגמאות:
  - > (pair? 1)
  - #f
  - > (pair? (cons 1 (cons 2 '())))
  - #t ;(cons 1 '(2))
  - > (pair? (cons 1 (cons 2 (cons 3 '()))))
  - #t ;(cons 1 '(2 3))
  - > (pair? (list 1 2))
  - #t ;(cons 1 '(2))

זוג

- > (pair? '(1 2))
- #t ;(cons 1 '(2))
- > (pair? ‘()) ; that’s null
- #f
- (pair? '('()))
- #t ;(cons '() null)
- (pair? '(1))
- #t ;(cons 1 '())

רשימות

- > (cons 1 (cons 2 (cons 3 null)))
- ‘(1 2 3) ;(cons 1 '(2 3))
- > (list 1 2 3)
- ‘(1 2 3) ;(cons 1 '(2 3))
- > (list 1 2 3 null)
- ‘(1 2 3 ()) ;(cons 1 '(2 3 ()))

- פונקציות נוספות:
- first – מחזיר את האיבר הראשון ברשימה.
- Rest – מחזיר רשימה ללא האבר הראשון.

דוגמא

- #lang pl
- #|
  A function *list-length* that takes as input a list,
  and outputs the number of elements in the list.
  |#

פתרון

- הסבר כללי על הפונקציה ופתרון מתמטי.

```
#|
Function list-length takes as argument a list (Listof Any).
The function returns the length of that list (Natural).
(list-length '()) = 0

(list-length '(x ...)) = 1 + (list-length '(...))
|#
```

פתרון

- הכרזה על הפונקציה, סוג הקלט וסוג הפלט.
- (: list-length : (Listof Any) -> Natural)
- הגדרת הפונקציה.
- (define (list-length list)
  (if (null? list)
  0
  (+ 1 (list-length (rest list)))))
- ביצוע בדיקות!
- (test (list-length '(1 2 3 4)) => 4)
  (test (list-length '(1 2 'a 'b true)) => 5)
  (test (list-length null) => 0)
  (test (list-length '()) => 0)

ההבדל בין רקורסיה ורקורסית זנב

- רקורסיה שליחה חוזרת עד לתנאי עצירה ולאחר מכן חישוב.
- נתונה הרקורסיה הבאה לחישוב עצרת:

```
(: fact : Natural -> Natural )
(define (fact n)
    (if (zero? n)
        1
         (* n (fact (- n 1)))))
```

ההבדל בין רקורסיה ורקורסית זנב

- השלבים לחישוב (fact 2) בעזרת פונקציה רקורסיבית:

- (* 2 (fact (- 2 1)))
  (* 2 (fact (1)))
  (* 2 (* 1 (fact (-1 1))))
  (* 2 (* 1 (fact (0))))
  (* 2 (* 1 1)))
  (* 2 1))
  2

ההבדל בין רקורסיה ורקורסית זנב

- רקורסית זנב חישוב ושליחה חוזרת עד לתנאי עצירה בדרך כלל רקורסית זנב דורשת פונקציית עזר:
- נתונה רקורסית הזנב הבאה לחישוב עצרת:

```
(: helper : Natural Natural -> Natural)
(define (helper n acc)
   (if (zero? n)
     acc
     (helper (- n 1) (* acc n))))
```

- פונקציית מעטפת:

```
(: fact : Natural -> Natural)
(define (fact n)
   (helper n 1))
```

ההבדל בין רקורסיה ורקורסית זנב

- השלבים לחישוב (fact 2) בעזרת רקורסית זנב:

```
(helper 2 1)
(helper (- 2 1) (* 1 2))
(helper 1 2)
(helper (- 1 1) (* 2 1))
(helper 0 2)
2            ;from helper
2            ;from fact
```

ההבדל בין רקורסיה ורקורסית זנב - דוגמא

- מציאת אורך של רשימה בעזרת פונקציה רקורסיבית:
-
```
(: list-length : ( Listof Any ) -> Natural )
( define ( list-length ls )
  (if ( null? ls )
      0
      (+ 1 ( list-length ( rest ls )))))
```

ההבדל בין רקורסיה ורקורסית זנב - דוגמא

- מציאת אורך של רשימה בעזרת רקורסית זנב:
-
```
(: list-length-tail : ( Listof Any ) -> Natural )
( define ( list-length-tail ls )
  (: helper-list-length-tail : Natural ( Listof Any ) -> Natural )
  ( define ( helper-list-length-tail acc ls )
    (if ( null? ls )
        acc
        ( helper-list-length-tail (+ 1 acc) ( rest ls ))))
  (helper-list-length-tail 0 ls))
```

ההבדל בין רקורסיה ורקורסית זנב - דוגמא

- חישוב סדרת פיבונאצ'י בעזרת פונקציה רקורסיבית:

```
(: fib : Integer -> Natural)
(define (fib n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [(>= n 2) (+ (fib (- n 1)) (fib (- n 2)))]
    [else (error 'fib "Expects Positive-Integer got ~s" n)]))
```

ההבדל בין רקורסיה ורקורסית זנב - דוגמא

- חישוב סדרת פיבונאצ'י בעזרת פונקציה רקורסיבית:

```
(: fib : Integer -> Natural)
(define (fib n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [(>= n 2) (+ (fib (- n 1)) (fib (- n 2)))]
    [else (error 'fib "Expects Positive-Integer got ~s" n)]))
```

- יש לציין שכאן התעוררה בעיה לגבי ה- (- n 2) לכן היה צורך לבחור ב Integer ולהוסיף את השגיאה כאשר הקלט הוא מספר שלילי.

ההבדל בין רקורסיה ורקורסית זנב - דוגמא

- חישוב סדרת פיבונאצ'י בעזרת רקורסית זנב:

```
(: fib-tail : Natural -> Natural)
(define (fib-tail n)
  (: fib-tail-help : Natural Natural Natural -> Natural)
  (define (fib-tail-help count f1 f2)
    (if (= n count)
        (+ f1 f2)
        (fib-tail-help (+ count 1) f2 (+ f1 f2))))
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [else (fib-tail-help 2 1 1)]))
```

- חשבו כמה קריאות רקורסיביות מתבצעות בכל אחד מהמימושים עבור (fib 5).

ההבדל בין רקורסיה ורקורסית זנב - דוגמא

- חישוב הסכום של האיברים ברשימת מספרים בעזרת פונקציה רקורסיבית:

```
(: sum-list : (Listof Number) -> Number)
(define (sum-list ls)
  (if (null? ls)
      0
      (+ (first ls) (sum-list (rest ls)))))
```

- חישוב הסכום של האיברים ברשימת מספרים בעזרת רקורסית זנב:

```
(: sum-list-tail : (Listof Number) -> Number)
(define (sum-list-tail ls)
  (: sum-list-tail-help : (Listof Number) Number -> Number)
  (define (sum-list-tail-help ls acc)
  (if (null? ls)
      acc
      (sum-list-tail-help (rest ls) (+ acc (first ls)))))
  (sum-list-tail-help ls 0))
```

שפות תכנות

תרגול 3

Let

הגדרת שם מזהה לבלוק לוקאלי.

```
(let ([id1 exp1] [id2 exp2].. [idn expn])
        body_using id1,id2 and idn )
```

```
> (let ([x 2][y 3]) (* x y))
6
> (let ([x 4]) (* x x))
16
> (let ([x 1] [y 2] [z 3]) (+ x y z))
6
```

Match

- (match value
  [pattern1 result-expr1]
  [pattern2 result-expr2]
  …)
- הפונקציה match מחזירה את הערך (result-expr) לתבנית הראשונה שמתאימה.
- תבנית שתמיד מצליחה:
- (match (list 1 2 3) [x x])
- '(1 2 3)

Match

- תבנית של סימבול:

- (let ([foo 'x])
  (match foo ['x "yes"] [else "no"]))

Match

- תבנית של סימבול:
- (let ([foo 'x])
  (match foo ['x "yes"] [else "no"]))
- יש לציין שבפונקציה match אין ייחודיות למילה else אלא המילה else היא כמו בדוגמא לעיל של תבנית שתמיד מצליחה ונקבל את else כמשתנה בעל ערך.
- ניתן להשתמש עם הערכים שלהם ביצענו קישור בתבנית, לדוגמא:
- (match '(1 2 3)
  [(list x y z) (+ x y z)])
- 6

דוגמא

- חישוב הסכום של האיברים ברשימת מספרים:
- (: sum : (Listof Number) -> Number)
```
(define (sum lst)
  (match lst
    ['() 0]
    [(cons h t) (+ h (sum t))]))
```
- השימוש בתבנית ה cons מאפשרת שימוש במקביל ב first ו rest של הרשימה, בדוגמא המשתנה h כבר מכיל את ה first והמשתנה t מכיל את ה rest.

Match

- תבנית של רשימה:
- (match '((1) (2) 3)
)] list (list x) (list y) z) (+ x y z([(
- 6
- בתבנית של רשימה ניתן להשתמש ב '...' המייצג חזרה (או לא) על התבנית הקודמת, לדוגמא:
- (match '((1 2) (3 4) (5 6) (7 8))
  [(list (list x y) ...) (append x y)])
- '(1 3 5 7 2 4 6 8) ; x => '(1 3 5 7) , y => '(2 4 6 8)

דוגמאות לתבניות

- (match value
  [id result-expr])
- תבנית המתאימה תמיד ו id מקושר.
- (match value
  [_ result-expr])
- תבנית המתאימה תמיד ללא קישור.
- (match value
  [(number : n) result-expr])
- תבנית המתאימה לכל מספר ו n מקושר.

דוגמאות לתבניות

- (match value
  [(symbol : s) result-expr])
- תבנית המתאימה לכל סימבול ו s מקושר.
- (match value
  [(string : s) result-expr])
- תבנית המתאימה לכל מחרוזת ו s מקושר.
- (match value
  [(sexpr : s) result-expr])
- תבנית המתאימה ל S- expressions ו s מקושר.

דוגמאות לתבניות

- (match value
  [(and pat1 pat2) result-expr])
- התאמה לשתי תבניות - 1pat וגם 2pat.
- (match value
  [(or pat1 pat2) result-expr])
- התאמה לאחת משתי תבניות - 1pat או 2pat.

דוגמאות

- (match '(1 a b c d)
)] list (number: n) (symbol :syms) ...)
  (list syms n)])
- '((a b c d) 1) ; syms => '(a b c d) , n => 1
- #lang pl untyped
  (match (list (list 'x 1 2 3) (list 'y 4 5))
  [(list (list (symbol: s) (number: n) ...) ...)
  (list s n)])
- '((x y) ((1 2 3) (4 5))) ; s => '(x y) , n=> '((1 2 3) (4 5))

דוגמאות

```
(define (foo x)
  (match x
    [(list (or 1 2 3)) 'single ]
    [(list (and x (list 1 _)) 2) x]
    [(or (list 1 x) (list 2 x)) x]))
```

- > (foo (list 3))
- 'single
- > (foo (list (list 1 99) 2))
- '(1 99)
- > (foo (list 1 10))
- 10
- > (foo (list 2 10))
- 10

דוגמאות

- פונקציה המקבלת רשימה ומחזירה רשימה הפוכה (האיבר שהיה ראשון ברשימה המקורית הופך להיות אחרון והאחרון ראשון וכו').
לדוגמא:

- (test (revr (list 1 2 3 4 5 6 7 8 9)) => '(9 8 7 6 5 4 3 2 1))
- (: revr : (Listof Any) => (Listof Any))
```
(define (revr ls)
  (match ls
    [(or (list) (list _)) ls]
    [(list bs ... a) (cons a (revr bs))]))
```

What dose the function do?

- (: fun1? : (Listof Symbol) -> Natural)
  (define (fun1? al)
    (match al
      [(cons 'x t) (+ 1 (fun1? t))]
      [(cons h t) (fun1? t)]
      [e 0]))
- (fun1? (list 'a 'x 'c 'd))
- 1

What dose the function do?

- (: fun2? : (Listof Number) -> (Listof Number))
```
(define (fun2? al)
  (: fun2-h : (Listof Number) (Listof Number) Number -> (Listof Number))
  (define (fun2-h ls1 ls2 acc)
    (match ls1
      [(cons f r) (let ([t (+ acc f)]) (fun2-h r (cons t ls2) t))]
      ['() (reverse ls2)]))
  (fun2-h al null 0))
```
- (fun2? (list 1 2 3 4 5 6))
- '(list 1 3 6 10 15 21)

שפות תכנות

תרגול 4

תרגילים

1. כתבו פונקציה המקבלת רשימה של מספרים ומחשבת את סכום האיברים ברשימה ע"י שימוש בפונקציה match.
2. כתבו פונקציה המקבלת רשימה והופכת את הרשימה מהסוף להתחלה.
3. כתבו פונקציה המקבלת רשימה של מספרים וסמלים המייצגת ביטוי של חיבור וחיסור מספרים ומחזירה את התוצאה של הביטוי.
לדוגמא: (calc-list '(1 + 2 - 3 + 4 + 5 - 6)) יחזיר 3. ניתן להניח שהביטוי תקין ומכיל רק + או -.

פתרונות לתרגילים

1. (: sum : (Listof Number) -> Number)
```
( define (sum lst)
        ( match lst
                [( cons h t) (+ h (sum t))]
                [ '() 0]) )
```
2. (: reverse-list : (Listof Any) -> (Listof Any))
```
(define (reverse-list lst)
        (match lst
                [(or (list) (list _)) lst]
                [(cons h t) (append (reverse-list t) (list h))]))
```

פתרונות לתרגילים

3.
```
(: calc-list : (Listof (U Number Symbol)) -> Number)
  (define (calc-list lst)
  (: help : (Listof (U Number Symbol)) Symbol -> Number)
        (define (help lst s)
              (match lst
                    ['() 0]
                    [(list (number: n) x ...)
                          (match s
                                ['+ (+ (help x s) n)]
                                ['- (- (help x s) n)])]
                    [(list (symbol: op) x ...) (help x op)]))
  (help lst '+))
```

Defining Data Types

- כדי להגדיר טיפוס חדש ב racket נשתמש בהצהרה:
- (define-type <name>
  [<variant1> <type> ...]
  [<variant2> <type> ...]
  [...])
- כאשר: name – שם הטיפוס החדש. וכל variant הוא שם של בנאי עם הטיפוסים המתאימים.
- דוגמא:
- (define-type Animal
  [Snake Symbol Number Symbol]
  [Tiger Symbol Number])

Defining Data Types

- לאחר הכרזה על הטיפוס, Racket מגדירה מספר פונקציות מובנות עבור הטיפוס.
- לדוגמא, לאחר הכרזה על Animal נוכל להשתמש בפונקציה Animal?:
- (Animal? (Tiger ‘Tony 12)
- #t
- (Animal? (Snake ‘Slimey 10 ‘rats)
- #t
- (Animal? (cons ‘Robi 19)
- #f

Defining Data Types - cases

- הפונקציה cases מאפשרת לבדוק עבור משתנה מסוג הטיפוס את ה variant המתאים לו ע"י תבניות בדומה לפונקציה match.
- דוגמא:
- (cases (Snake 'Slimey 10 'rats)
  [(Snake n w f) n]
  [(Tiger n sc) n])
- 'Slimey
- ב cases יש לכסות את כל ה variants של הטיפוס, אחרת נקבל שגיאה.
- בcases אין צורך ב-else, משום שאם קיבלנו משתנה מסוג הטיפוס, הוא חייב להיות מתאים לתבנית של אחד מה-variants.
- לדוגמא:
- (cases (Snake 'Slimey 10 'rats)
  [(Snake n w f) n])
- נקבל שגיאה.

תרגילים

1. כתבו טיפוס בשם: Person עם ה variants הבאים: Teacher – שלו יש שם (string), ותק (מספר טבעי), ומשכורת (כמספר שלם). Student – שלו יש שם (string), שנת לימודים וממוצע ציונים (מספר ממשי).
כתבו פונקציה המקבלת רשימה של Person ומחזירה רשימה ובה יש 2 איברים: מספר המורים ומספר הסטודנטים.

2. כתבו טיפוס בשם MyString המייצג מחרוזת של תווים עם ה variants: Empty – המייצג מחרוזת ריקה. STR המורכב מתו ראשון ומהטיפוס MyString – המשך המחרוזת.
כתבו פונקציה המקבלת MyString ומחזירה את אורך המחרוזת, השתמשו ב cases.

3. כתבו טיפוס בשם MyList המייצג רשימה עם מספרים או סימבולים, עם ה variantsהבאים :
   1. Empty – מייצג רשימה ריקה.
   2. Num-node המורכב מ Number ומהטיפוס MyList – המשך המחרוזת.
   3. Sym-node המורכב מ Symbol ומהטיפוס MyList – המשך המחרוזת.
   4. L-node. המורכב משתי רשימות מהטיפוס שיצרתם- MyList.

   כתבו פונקציה הבודקת כמה אברים מסוג Symbol יש ברשימה.

פתרונות לתרגילים

1. (define-type Person
   [Student String Natural Real]
   [Teacher String Natural Integer])

```
(: Person-Number : (Listof Person) -> (Listof Natural))
(define (Person-Number lst)
  (Person-Number-h lst 0 0))
(: Person-Number-h : (Listof Person) Natural Natural -> (Listof Natural))
(define (Person-Number-h lst tn sn)
  (if (null? lst)
      (list tn sn)
      (cases (first lst)
        [(Student x y z) (Person-Number-h (rest lst) tn (+ sn 1))]
        [(Teacher x y z) (Person-Number-h (rest lst) (+ tn 1) sn)])))
```

פתרונות לתרגילים

2. (define-type MyString
   [Empty]
   [STR Char MyString])

```
(STR #\H (STR #\e (STR #\l (STR #\l (Empty)))))
(: MyStringLen : MyString -> Natural)
(define (MyStringLen str)
  (cases str
    [(Empty) 0]
    [(STR c s) (+ 1 (MyStringLen s))]))
```

פתרונות לתרגילים

3.

```
(define-type MyList
  [Empty]
  [Num-node Number MyList]
  [Sym-node Symbol MyList]
   [L-node MyList MyList])

(: countSyms : MyList -> Natural)
(define (countSyms L)
 (cases L
  [(Empty) 0]
  [(Sym-node _ rest) (add1 (countSyms rest))]
  [(Num-node _ rest) (countSyms rest)]
  [(L-node lst1 lst2) (+ (countSyms lst1) (countSyms lst2))]))

(countSyms (Sym-node 'w (Sym-node 'q (Num-node 7 (Sym-node 'p (L-node (Empty) (Empty)))))))
  3
(countSyms (Sym-node 'w (Sym-node 'q (Num-node 7 (Sym-node 'p (L-node (Sym-node 'o (Empty)) (Empty)))))))
4
```

שפות תכנות

תרגול 5

כתיבת parser – שפת הרגיסטרים (ROL)

- בתרגיל זה נכתוב דקדוק ו parser עבור שפת הרגיסטרים המאפשרת פעולות לוגיות על רגיסטרים (שהם למעשה סדרות של אפסים ואחדות).
- תיאור השפה: כל ביטוי בשפה הוא מהצורה:
- {reg-len= len A}
- כאשר len הוא מספר שלם, A הוא ביטוי המתאר סדרת פעולות על רגיסטרים. ביטוי כזה מקיים את הכללים הבאים:
  - סדרה של אפסים ואחדות – עטופים בסוגריים מסולסלים (המייצגת ערך של רגיסטר) היא חוקית כסדרת פעולות על רגיסטרים.
  - אם B,A סדרות פעולות על רגיסטרים, אז גם הביטוי המתקבל ע"י שימוש באופרטור and או אופרטור or כאשר A, B הם האופרנדים, ועטיפת הביטוי כולו בסוגריים מסולסלים – הוא חוקי כסדרת פעולות על רגיסטרים. גם הביטוי המתקבל ע"י שימוש באופרטור shl כאשר A הוא האופרנד, ועטיפת הביטוי כולו בסוגריים מסולסלים – הוא חוקי כסדרת פעולות על רגיסטרים.

כתיבת parser – שפת הרגיסטרים (ROL)

- הביטויים הבאים הם ביטויים חוקיים בשפה:
  - {reg-len= 4 {1 0 0 0}}
  - {reg-len= 4 {shl {1 0 0 0}}}
  - {reg-len= 4 {and {shl {1 0 1 0}} {shl {1 0 1 0}}}}
  - {reg-len= 4 {or {and {shl {1 0 1 0}} {shl {1 0 1 0}}} {1 0 1 0}}}

כתיבת parser – שפת הרגיסטרים (ROL)

- כתבו BNF עבור השפה ROL. השתמשו ב <num> עבור כל המספרים בדומה לדקדוק של AE.
- כתבו את הטיפוס ROL המתאים לשפה שהגדרנו.
- כתבו את ה parser עבור השפה, כלומר ממשו את parse-sexpr המקבלת Sexpr ומחזירה ROL.

BNF

- <ROL> ::= {reg-len = <num> <RegE> }
- <RegE> ::=<Bits>

  |{and <RegE> <RegE>}

  |{or <RegE> <RegE>}

  |{shl <RegE>}

- <Bits> ::= <bit>

  |<bit> <Bits>

- <bit>::= 1 | 0

define-type

- (define-type BIT = (U 0 1))

  (define-type Bit-List = (Listof BIT))

```
(define-type RegE
    [Reg Bit-List]
    [And RegE RegE]
    [Or RegE RegE]
    [Shl RegE])
```

How to parse bit list?

```
(: list->bit-list : (Listof Any) -> Bit-List)
    (define (list->bit-list lst)
        (cond
            [(null? lst) null]
            [(eq? (first lst) 1)
                (cons 1 (list->bit-list (rest lst)))]
            [else (cons 0 (list->bit-list (rest lst)))]))
```

parse-sexpr-RegL

```
(: parse-sexpr-RegL : Sexpr Number -> RegE)
(define (parse-sexpr-RegL sexpr len)
        (match sexpr
        [(list (and a (or 1 0)) ... )
                (if (= len (length a))
                        (Reg (list->bit-list a))
                        (error 'parse-sexpr-RegE "wrong number of bits in ~s" a)) ]
        [(list 'and list1 list2) (And (parse-sexpr-RegL list1 len) (parse-sexpr-RegL list2 len))]
        [(list 'or list1 list2)     (Or (parse-sexpr-RegL list1 len) (parse-sexpr-RegL list2 len))]
        [(list 'shl list)           (Shl (parse-sexpr-RegL list len))]
[else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

parse-sexpr

```
(: parse-sexpr : Sexpr -> RegE)
(define (parse-sexpr sexpr)
    (match sexpr
        [(list 'reg-len ‘=      (number: n)    args)
            (if (> n 0)
                (parse-sexpr-RegL args n)
                (error 'parse-sexpr "Register length must be at least 1 ~s" sexpr) )]
        [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

Parse one of the following:

{reg-len= 4 {shl {1 0 0 0}}}

(Shl (Reg '(1 0 0 0)))

{reg-len= 4 {and {shl {1 0 1 0}} {shl {1 0 1 0}}}}

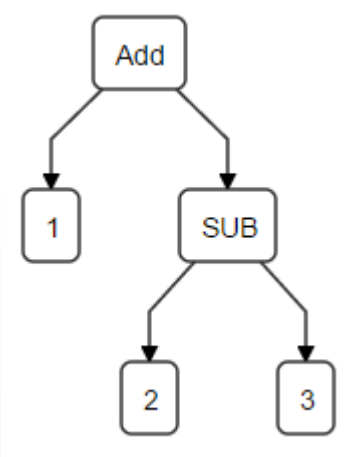
(And (Shl (Reg '(1 0 1 0))) (Shl (Reg '(1 0 1 0))))

שפות תכנות

תרגול 6

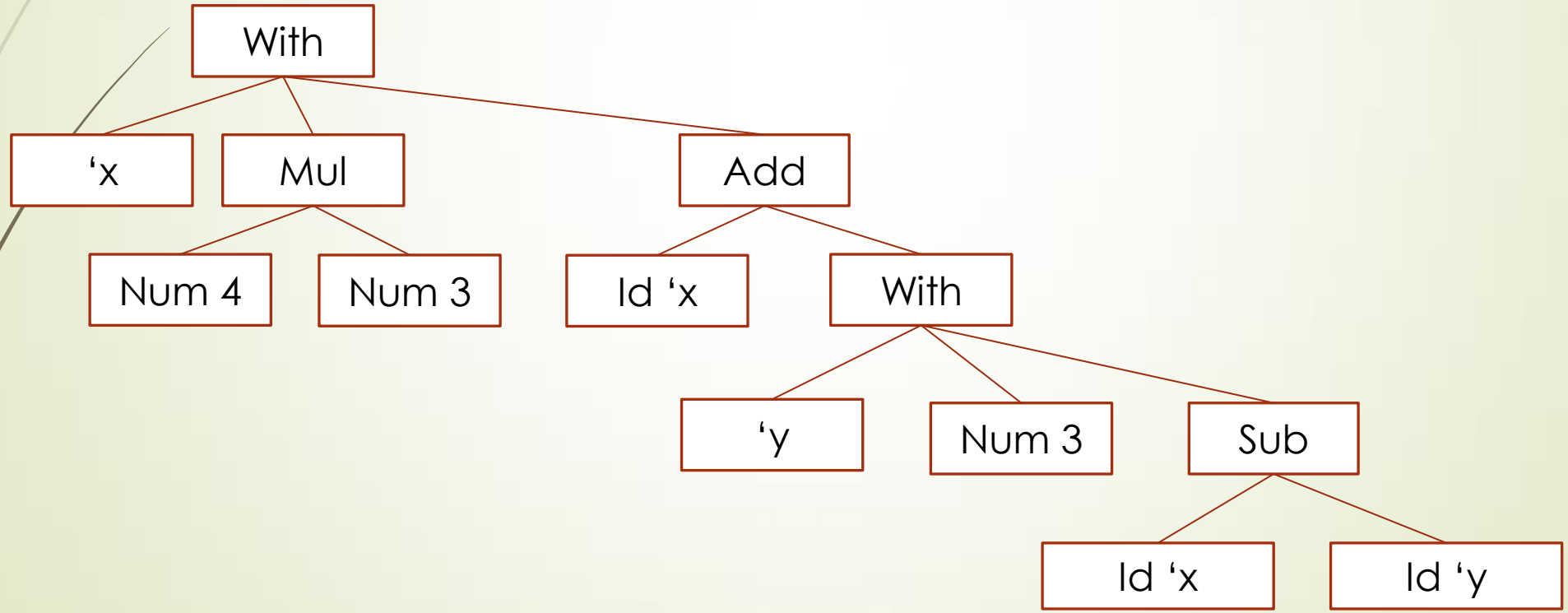
Abstract Syntax Tree (AST)

- הגדרה: AST הוא עץ המייצג את המבנה התחבירי של הקוד.
- העץ מיוצר ע"י פעולת parser על הקוד.
- דוגמא: עץ עבור הקוד הבא בשפה WAE:
  - {+ 1 {- 2 3}}



Abstract Syntax Tree (AST)

- תרגיל: בנו AST עבור הקודים הבאים בשפת WAE, בעזרת parse.
  - {with {x {* 4 3}} {+ x {with {y 3} {- x y}}}}



כתיבת parser – הרחבת WAE

- בשיעור ראיתם את ה parser לשפה WAE.
- נרצה להרחיב את השפה ולאפשר שימוש בביטויים נוספים:
- נוסיף את האופרטורים - ^ , sqrt, max. כך שהביטויים הבאים חוקיים בשפה:
  - {^ 2 {+ 5 8}}
  - {sqrt 4}
  - {max <3> {sqrt 5} {* 1 3} 6}
  - {+ {max <1> 5} {sqrt {max <5> 1 2 3 4 5} }

כתיבת parser – הרחבת WAE

- שלב 1: הרחיבו את הדקדוק כך שיתאים לשפה החדשה.
- שלב 2: הרחיבו את הטיפוס WAE בהתאם, הוסיפו את הקוד הנדרש.
- שלב 3: כתבו parser לשפה חדשה – הרחיבו את הפונקציה parse-sexpr בהתאם.

כתיבת parser – הרחבת WAE - פתרון

- הדקדוק:

- <EWAE> ::= <num>
  | {+ <EWAE> <EWAE>}
  | {- <EWAE> <EWAE>}
  | {* <EWAE> <EWAE>}
  | {/ <EWAE> <EWAE>}
- | {^ <EWAE> <EWAE>}
  | {sqrt <EWAE>}
  | {max <<num>> <VALUES>}
  | {with {<id> <EWAE>} <EWAE>}
  | <id>
- <VALUES> ::= <EWAE> | <EWAE> <VALUES>

כתיבת parser – הרחבת WAE - פתרון

- הטיפוס:

```
(define-type EWAE
[Num Number]
  [Add EWAE EWAE]
  [Sub EWAE EWAE]
  [Mul EWAE EWAE]
  [Div EWAE EWAE]
[Pow EWAE EWAE]
[Sqrt EWAE]
[Max Number (Listof  EWAE)]
[Id Symbol]
[With Symbol EWAE EWAE])
```

כתיבת parser – הרחבת WAE - פתרון

- ה parser:

```
(: parse-sexpr : Sexpr-> EWAE)
(define (parse-sexpr sexpr)
  (match sexpr
        [(number: n) (Num n)]
        [(symbol: name) (Id name)]
        [(cons 'with more)
              (match sexpr
                    [(list 'with (list (symbol: name) named) body)
                    (With name (parse-sexpr named) (parse-sexpr body))]
                    [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])])
```

כתיבת parser – הרחבת WAE - פתרון

```
[(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
[(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
[(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
[(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
[(list ‘^ lhs rhs) (Pow (parse-sexpr lhs) (parse-sexpr rhs))]
[(list ‘sqrt arg) (Sqrt (parse-sexpr arg))]
[(list ‘max ‘< (number: n) ‘> args …) (Max n (parse-sexpr* args))]
 [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

כתיבת parser – הרחבת WAE - פתרון

- (: parse : String -> EWAE)
  (define (parse str)
  (parse-sexpr (string->sexpr str)))

- (: parse-sexpr* : (Listof Sexpr)-> (Listof EWAE))
  (define (parse-sexpr* lst)
  (if (null? lst)
  null
  (append (list (parse-sexpr (first lst)) (parse-sexpr* (rest lst)))))

Abstract Syntax Tree (AST)

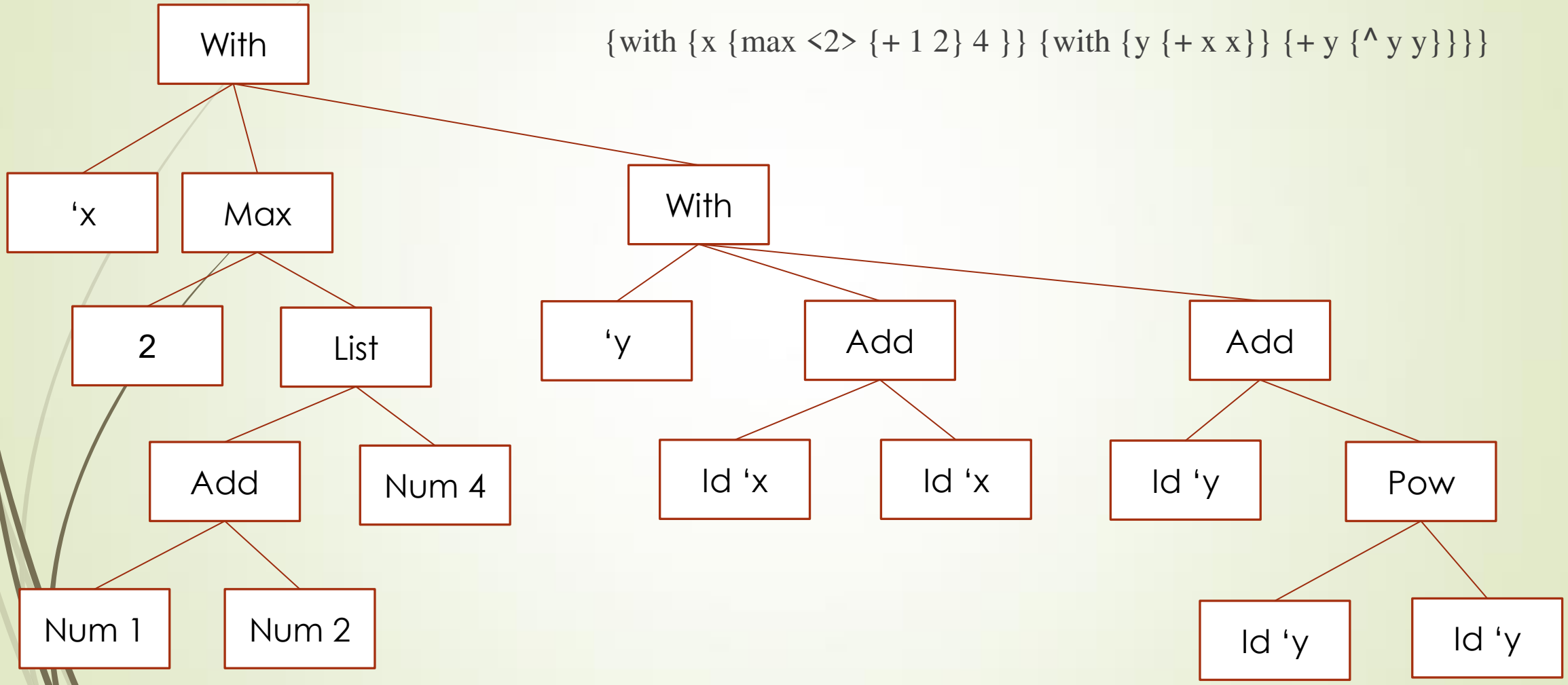
- לסיכום

בנו AST עבור הקודים הבאים בשפת WAE, בעזרת parse.

- {with {x {max <2> {+ 1 2} 4 }} {with {y {+ x x}} {+ y {^ y y}}}}

Abstract Syntax Tree (AST)

{with {x {max <2> {+ 1 2} 4 }} {with {y {+ x x}} {+ y {^ y y}}}}



שפות תכנות

תרגול 7

BNF for the WAE language

```
#|
    <WAE> ::= <num>
            | { + <WAE> <WAE> }
            | { - <WAE> <WAE> }
            | { * <WAE> <WAE> }
            | { / <WAE> <WAE> }
            | { with { <id> <WAE> } <WAE> }
            | <id>
|#
```

WAE abstract syntax trees

```
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])
```

Evaluation rules:

```
#| Formal specs for `subst':
   (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>,
   `y' is a *different* <id>)
      N[v/x]                  = N
      {+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]          = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]          = {/ E1[v/x] E2[v/x]}
      y[v/x]                  = y
      x[v/x]                  = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
|#
```

subst

```
(: subst : WAE Symbol WAE -> WAE)
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
```

subst

```
[(Id name) (if (eq? name from) to expr)]
[(With bound-id named-expr bound-body)
  (With bound-id
        (subst named-expr from to)
        (if (eq? bound-id from)
          bound-body
          (subst bound-body from to)))]))
```

Flow the subst code

- {with {x {* 4 3}} {+ x {with {y 3} {- x y}}}}
- {with {x {* 4 3}} {+ x {with {x 3} {- x x}}}}
- What will be different?

- {with {x {* 4 3}} {+ x {with {y 3} {- z y}}}}

and now what?

containsFreeInstance

```
(: containsFreeInstance? : WAE -> Boolean)

 (define (containsFreeInstance? expr)
  (cases expr
          [(Num n) #f]
          [(Add l r) (or (containsFreeInstance? l) (containsFreeInstance? r))]
          [(Sub l r) (or (containsFreeInstance? l) (containsFreeInstance? r))]
          [(Mul l r) (or (containsFreeInstance? l) (containsFreeInstance? r))]
          [(Div l r) (or (containsFreeInstance? l) (containsFreeInstance? r))]
```

containsFreeInstance

```
[(With bound-id named-expr bound-body)
    (or (containsFreeInstance? named-expr)
        (containsFreeInstance? (subst bound-body
                                      bound-id
                                      (Num 0))))]
[(Id name) #t]))
```

הרחבת - WAE הוספת עצרת.

```
#| BNF for the WAE language:

<WAE> ::= <num>
  | { + <WAE> <WAE> }
  | { - <WAE> <WAE> }
  | { * <WAE> <WAE> }
  | { / <WAE> <WAE> }
  | { with { <id> <WAE> } <WAE> }
  | <id>
  | ????

|#
```

שלב ראשון – הרחבת BNF

נרצה להוסיף ל WAE את היכולת לקבל אופרטור חד מקומי '!' (עצרת).

ראשית נדאג לעדכן את ה- BNF.

דוגמאות:

- {! 2} -> 2
- {! {+ 3 4}} -> 5040
- {! {+ {- 1 2} 3}} -> 2
- {+ {! 8} 5 } -> 40325

הרחבת - WAE הוספת עצרת.

```
#| BNF for the WAE language:

<WAE> ::= <num>
  | { + <WAE> <WAE> }
  | { - <WAE> <WAE> }
  | { * <WAE> <WAE> }
  | { / <WAE> <WAE> }
  | { with { <id> <WAE> } <WAE> }
  | <id>
  | ????

|#
```

שלב ראשון – הרחבת BNF

נרצה להוסיף ל WAE את היכולת לקבל אופרטור חד מקומי '!' (עצרת).

ראשית נדאג לעדכן את ה- BNF.

דוגמאות:

- {! 2} -> 2
- {! {+ 3 4}} -> 5040
- {! {+ {- 1 2} 3}} -> 2
- {+ {! 8} 5 } -> 40325

הרחבת - WAE הוספת עצרת.

```
#| BNF for the WAE language:

<WAE> ::= <num>
  | { + <WAE> <WAE> }
  | { - <WAE> <WAE> }
  | { * <WAE> <WAE> }
  | { / <WAE> <WAE> }
  | { with { <id> <WAE> } <WAE> }
  | <id>
  | {! <WAE>}

|#
```

שלב ראשון – הרחבת BNF

נרצה להוסיף ל WAE את היכולת לקבל אופרטור חד מקומי '!' (עצרת).

ראשית נדאג לעדכן את ה- BNF.

דוגמאות:

- {! 2} -> 2
- {! {+ 3 4}} -> 5040
- {! {+ {- 1 2} 3}} -> 2
- {+ {! 8} 5 } -> 40325

הרחבת - WAE הוספת עצרת.

שלב שני – הרחבת ה-parser

בכדי להרחיב את ה parser ובהמשך גם את eval נגדיר constractor חדש מסוג WAE:

```
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE]
  ????
)
```

:

הרחבת - WAE הוספת עצרת.

שלב שני – הרחבת ה-parser

בכדי להרחיב את ה parser ובהמשך גם את eval נגדיר constractor חדש מסוג WAE:

```
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE]
  [Fact WAE]
)
```

:

הרחבת - WAE הוספת עצרת.

שלב שני – הרחבת ה-parser

```
(: parse-sexpr : Sexpr -> WAE)
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n) (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
      (match sexpr
          [(list 'with (list (symbol: name) named) body)
                          (With name (parse-sexpr named) (parse-sexpr body))]
          [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div  (parse-sexpr lhs) (parse-sexpr rhs))]
      ?????????
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

הרחבת - WAE הוספת עצרת.

שלב שני – הרחבת ה-parser

```
(: parse-sexpr : Sexpr -> WAE)
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n) (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
      (match sexpr
          [(list 'with (list (symbol: name) named) body)
                          (With name (parse-sexpr named) (parse-sexpr body))]
          [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '! exp)     (Fact (parse-sexpr exp))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

לאחר שסיימנו לטפל בparser, נשאר לנו להרחיב את eval ואת subst כך שנוכל לחשב את הביטויים האלה.

- עלינו לממש את הפונקציה fact.
- עלינו להוסיף קריאה לfact בeval.
- עלינו להוסיף טיפול בfact ובsubst.

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

מימוש הפונקציה :fact

- נשתמש במימוש מהתרגול השני.
- נוסיף בדיקת קלט.

```
(: fact-op : Number -> Natural)
(define (fact-op n) (: fact-op-help : Natural Natural -> Natural)
    (define (fact-op-help n acc)
        (if (> n 0)
            (fact-op-help (- n 1) (* n acc))
            acc))
    (if (and (integer? n) (exact? n) (not (negative? n)))
        (fact-op-help n 1)
        (error 'fact-op "Expected Exact-Nonnegative-Integer got: ~s" n)))
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה eval.
  - **פורמלית (specifications).**
  - תכנותית.

```
#| Formal specs for `eval':
      eval(N)           = N
      eval({+ E1 E2}) = eval(E1) + eval(E2)
      eval({- E1 E2}) = eval(E1) - eval(E2)
      eval({* E1 E2}) = eval(E1) * eval(E2)
      eval({/ E1 E2}) = eval(E1) / eval(E2)
      eval(id)          = error!
      eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
      ??????
|#
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה eval.
  - **פורמלית (specifications).**
  - תכנותית.

```
#| Formal specs for `eval':
      eval(N)           = N
      eval({+ E1 E2}) = eval(E1) + eval(E2)
      eval({- E1 E2}) = eval(E1) - eval(E2)
      eval({* E1 E2}) = eval(E1) * eval(E2)
      eval({/ E1 E2}) = eval(E1) / eval(E2)
      eval(id)          = error!
      eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
      eval({! E1})      = eval(E1)!
|#
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה eval.
  - פורמלית (specifications).
  - **תכנותית.**

```
(: eval : WAE -> Number)
(define (eval expr) (cases expr
     [(Num n) n]
     [(Add l r) (+ (eval l) (eval r))]
     [(Sub l r) (- (eval l) (eval r))]
     [(Mul l r) (* (eval l) (eval r))]
     [(Div l r) (/ (eval l) (eval r))]
      ?????
     [(With bound-id named-expr bound-body)
      (eval (subst bound-body bound-id (Num (eval named-expr))))]
     [(Id name) (error 'eval "free identifier: ~s" name)]))
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה eval.
  - פורמלית (specifications).
  - **תכנותית.**

```
(: eval : WAE -> Number)
(define (eval expr) (cases expr
     [(Num n) n]
     [(Add l r) (+ (eval l) (eval r))]
     [(Sub l r) (- (eval l) (eval r))]
     [(Mul l r) (* (eval l) (eval r))]
     [(Div l r) (/ (eval l) (eval r))]
     [(Fact n ) (fact-op (eval n))]
     [(With bound-id named-expr bound-body)
      (eval (subst bound-body bound-id (Num (eval named-expr))))]
     [(Id name) (error 'eval "free identifier: ~s" name)]))
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה subst.
  - **פורמלית (specifications).**
  - תכנותית.

```
#| Formal specs for `subst':
    (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>,
`y' is a *different* <id>)
      N[v/x]                  = N
      {+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]          = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]          = {/ E1[v/x] E2[v/x]}
      y[v/x]                  = y
      x[v/x]                  = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
        ?????
|#
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה subst.
  - **פורמלית (specifications).**
  - תכנותית.

```
#| Formal specs for `subst':
    (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>,
`y' is a *different* <id>)
      N[v/x]                  = N
      {+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]          = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]          = {/ E1[v/x] E2[v/x]}
      y[v/x]                  = y
      x[v/x]                  = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
      {! E1}[v/x]             = {! E1[v/x]}
|#
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה subst.
  - פורמלית (specifications).
  - **תכנותית.**

```
(: subst : WAE Symbol WAE -> WAE)
(define (subst expr from to)
(cases expr
     [(Num n) expr]
     [(Add l r) (Add (subst l from to) (subst r from to))]
     [(Sub l r) (Sub (subst l from to) (subst r from to))]
     [(Mul l r) (Mul (subst l from to) (subst r from to))]
     [(Div l r) (Div (subst l from to) (subst r from to))]
      ????
     [(Id name) (if (eq? name from) to expr)]
     [(With bound-id named-expr bound-body)
          (With bound-id
            (subst named-expr from to)
            (if (eq? bound-id from) bound-body
                (subst bound-body from to)))]))
```

הרחבת - WAE הוספת עצרת.

שלב שלישי – הרחבת eval.

- נעדכן את הפרוצדורה subst.
  - פורמלית (specifications).
  - **תכנותית.**

```
(: subst : WAE Symbol WAE -> WAE)
(define (subst expr from to)
(cases expr
     [(Num n) expr]
     [(Add l r) (Add (subst l from to) (subst r from to))]
     [(Sub l r) (Sub (subst l from to) (subst r from to))]
     [(Mul l r) (Mul (subst l from to) (subst r from to))]
     [(Div l r) (Div (subst l from to) (subst r from to))]
     [(Fact n) (Fact (subst n from to))]
     [(Id name) (if (eq? name from) to expr)]
     [(With bound-id named-expr bound-body)
          (With bound-id
            (subst named-expr from to)
            (if (eq? bound-id from) bound-body
                (subst bound-body from to)))]))
```

חשבו את הביטוי הבא:

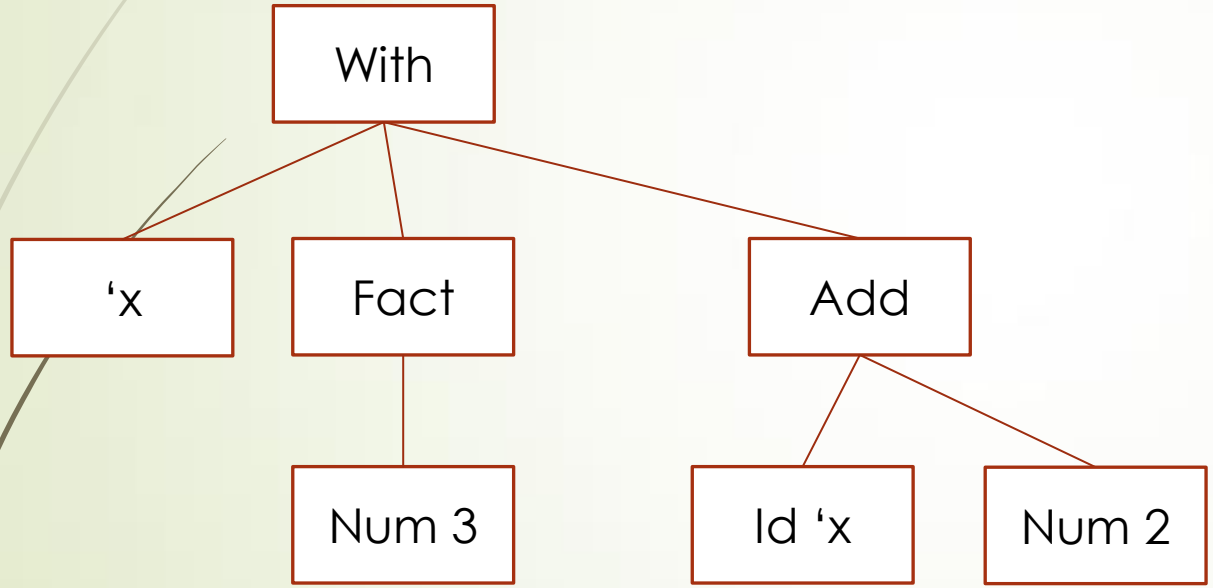
```
(run "{with {x {! 3 }} {+ x 2}}")

(run "{with {x {- 5 2}} {with {y {! x}} {- x y}}}")
```

- הראו את חישוב parse (בנו AST).
- הראו את חישוב eval.

חשבו את הביטוי הבא:

```
(run "{with {x {! 3 }} {+ x 2}}")
```

- הראו את חישוב parse (בנו AST).



חשבו את הביטוי הבא:

```
(run "{with {x {! 3 }} {+ x 2}}")
```

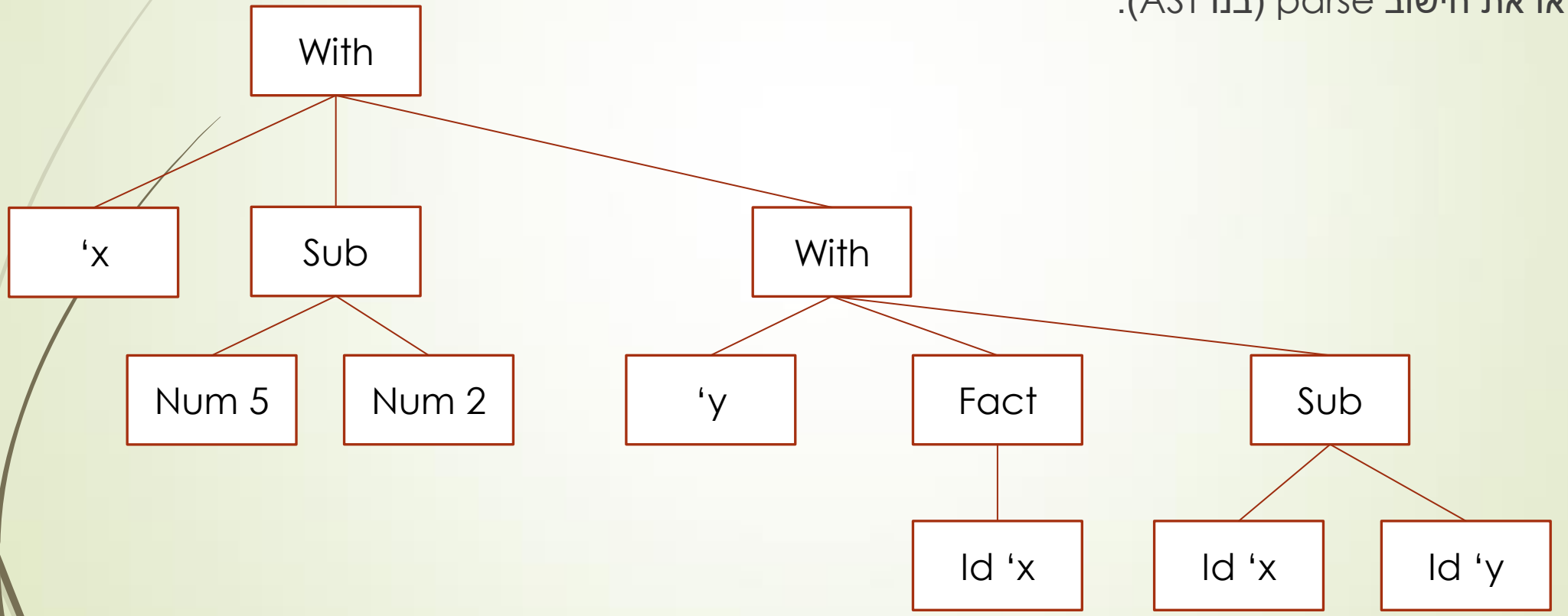
- הראו את חישוב eval.

```
eval1:(With ('x (Fact (Num 3))) (Add (Id 'x) (Num 2)))
res1:  8
eval2:(Fact (Num 3))
res2:  6
eval3:(Num 3)
res3:  3
eval4:(Add (Num 6) (Num 2))
res4:  8
eval5:(Num 6)
res5:  6
eval6:(Num 2)
res6:  2
```

חשבו את הביטוי הבא:

```
(run "{with {x {- 5 2}} {with {y {! x}} {- x y}}}")
```

- הראו את חישוב parse (בנו AST).



חשבו את הביטוי הבא:

```
(run "{with {x {- 5 2}} {with {y {! x}} {- x y}}}")
```

- הראו את חישוב eval.

```
eval1:(With ('x (Sub (Num 5)(Num 2))) (With ('y (Fact (Id 'x))) (Sub (Id 'x) (Id 'y))))
res1:  -3
eval2:(Sub (Num 5)(Num 2))
res2:  3
eval3:(Num 5)
res3:  5
eval4:(Num 2)
res4:  2
eval5:(With ('y (Fact (Num 3))) (Sub (Num 3) (Id 'y)))
res5:  -3
eval6:(Fact (Num 3))
res6:  6
eval7:(Num 3)
res7:  3
```

```
eval8:(Sub (Num 3) (Num 6))
res8:  -3
eval9:(Num 3)
res9:  3
eval10:(Num 6)
res10:  6
```

שפות תכנות תרגול 8

FLANG

הוספת Fun ו- Call

שלב ראשון – הרחבת BNF

```
# |  <FLANG> ::= <num>
                | { + <FLANG> <FLANG> }
                | { - <FLANG> <FLANG> }
                | { * <FLANG> <FLANG> }
                | { / <FLANG> <FLANG> }
                | { with { <id> <FLANG> } <FLANG> }
                |  <id>
                | { fun { <id> } <FLANG> }
                | { call <FLANG> <FLANG> }

| #
```

הוספת Fun ו- Call

שלב שני – הרחבת ה-parse
constractors

```
(define-type FLANG
    [Num Number]
    [Add FLANG FLANG]
    [Sub FLANG FLANG]
    [Mul FLANG FLANG]
    [Div FLANG FLANG]
    [Id Symbol]
    [With Symbol FLANG FLANG]
    [Fun Symbol FLANG]
    [Call FLANG FLANG])
```

הוספת Fun ו- Call

שלב שני – הרחבת ה-parse

parse

```
(: parse-sexpr : Sexpr -> FLANG)
        ;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
        (match sexpr
                [(number: n) (Num n)]
                [(symbol: name) (Id name)]
                [(cons 'with more)
                  (match sexpr
                        [(list 'with (list (symbol: name) named) body)
                           (With name (parse-sexpr named) (parse-sexpr body))]
                        [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
               [(cons 'fun more)
                        (match sexpr
                                [(list 'fun (list (symbol: name)) body)
                                   (Fun name (parse-sexpr body))]
                                [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
```

הוספת Fun ו- Call

שלב שני – הרחבת ה-parse

parse

```
[(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
[(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
[(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
[(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
[(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
[else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

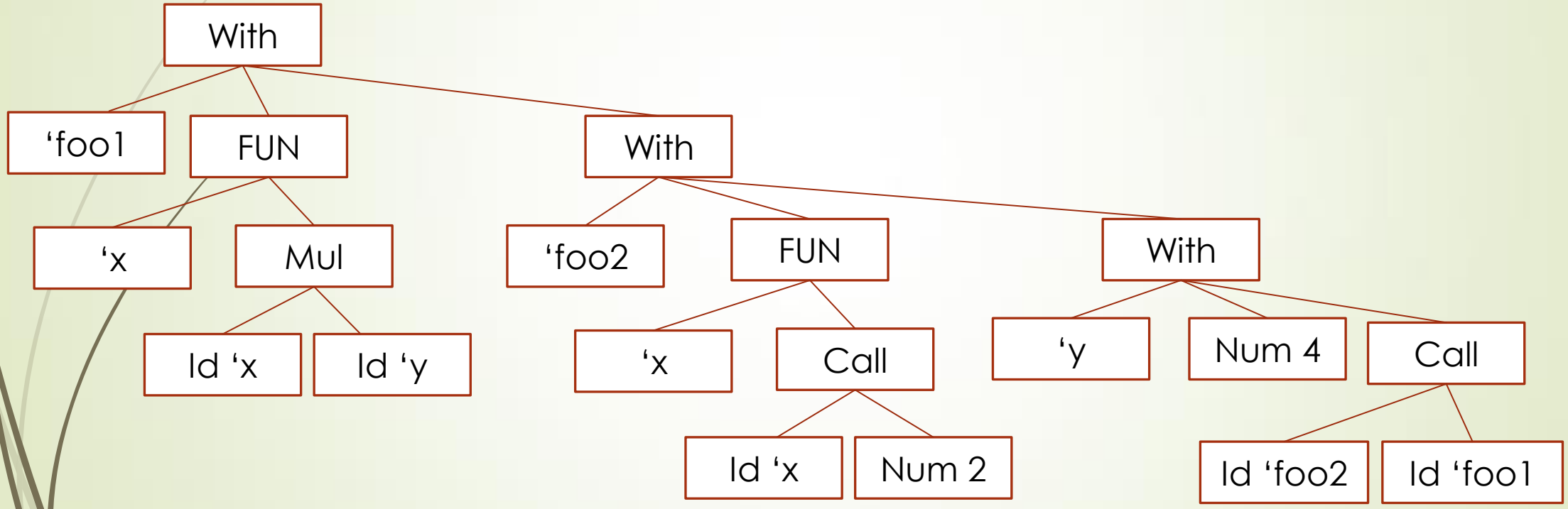
```
(: parse : String -> FLANG)
;; parses a string containing a FLANG expression to a FLANG AST
(define (parse str)
        (parse-sexpr (string->sexpr str)))
```

תרגיל 1

- ציירו את עץ התחביר האבסטרקטי המתאר את הביטוי הנתון במרכאות (כלומר את התוצאה של הפעלת parse על ביטוי זה). השתמשו ב FLANG

```
(run "{with {foo1 {fun {x} {* x y}}}
        {with {foo2 {fun {x} {call x 2}}}
          {with {y 4}
            {call foo2 foo1}}}}")
```

```
(run "{with {foo1 {fun {x} {* x y}}}
        {with {foo2 {fun {x} {call x 2}}}
          {with {y 4}
            {call foo2 foo1}}}}")
```



הוספת Fun ו- Call

שלב שלישי – הרחבת ה-eval

subst

```
# | Subst rules:

N[v/x]                  = N
{+ E1 E2}[v/x]          = {+ E1[v/x] E2[v/x]}
{- E1 E2}[v/x]          = {- E1[v/x] E2[v/x]}
{* E1 E2}[v/x]          = {* E1[v/x] E2[v/x]}
{/ E1 E2}[v/x]          = {/ E1[v/x] E2[v/x]}
y[v/x]                  = y
x[v/x]                  = v
{with {y E1} E2}[v/x]   = {with {y E1[v/x]} E2[v/x]} ; if y =/= x
{with {x E1} E2}[v/x]   = {with {x E1[v/x]} E2}
{call E1 E2}[v/x]       = {call E1[v/x] E2[v/x]}
{fun {y} E}[v/x]        = {fun {y} E[v/x]}      ; if y =/= x
{fun {x} E}[v/x]        = {fun {x} E}

| #
```

הוספת Fun ו- Call

שלב שלישי – הרחבת ה-eval

subst

```
(: subst : FLANG Symbol FLANG -> FLANG)
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
      (With bound-id
            (subst named-expr from to)
            (if (eq? bound-id from) bound-body
                (subst bound-body from to)))]))
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
      (if (eq? bound-id from)
            expr
            (Fun bound-id (subst bound-body from to)))]))
```

הוספת Fun ו- Call

שלב שלישי – הרחבת ה-eval

subst

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
      (: Num->number : FLANG -> Number)
      (define (Num->number e)
         (cases e
               [(Num n) n]
               [else (error 'arith-op "expects a number, got: ~s" e)]))
      (Num (op (Num->number expr1) (Num->number expr2))))
```

הוספת Fun ו- Call

שלב שלישי – הרחבת ה-eval

eval

```
# |  eval rules:

      eval(N) = N
      eval({+ E1 E2})          = eval(E1) + eval(E2)
      eval({- E1 E2})          = eval(E1) - eval(E2)
      eval({* E1 E2})          = eval(E1) * eval(E2)
      eval({/ E1 E2})          = eval(E1) / eval(E2)
      eval(id)                 = error!
      eval({with {x E1} E2})   = eval(E2[eval(E1)/x])
      eval(FUN)                = FUN          ; assuming FUN is a function expression
      eval({call E1 E2})       = eval(Ef[eval(E2)/x])   if eval(E1)={fun {x} Ef}
                               = error!                 otherwise

| #
```

הוספת Fun ו- Call

שלב שלישי – הרחבת ה-eval

eval

```
(: eval : FLANG -> FLANG)
;; evaluates FLANG expressions by reducing them to *expressions*
(define (eval expr)
   (cases expr
      [(Num n) expr]
      [(Add l r) (arith-op + (eval l) (eval r))]
      [(Sub l r) (arith-op - (eval l) (eval r))]
      [(Mul l r) (arith-op * (eval l) (eval r))]
      [(Div l r) (arith-op / (eval l) (eval r))]
      [(With bound-id named-expr bound-body)
        (eval (subst bound-body bound-id (eval named-expr)))]
      [(Id name) (error 'eval "free identifier: ~s" name)]
      [(Fun bound-id bound-body) expr]
      [(Call fun-expr arg-expr)
         (let ([fval (eval fun-expr)])
            (cases fval
                  [(Fun bound-id bound-body) (eval (subst bound-body bound-id (eval arg-expr)))]
                  [else (error 'eval "`call' expects a function, got: ~s" fval)]))]))
```

הוספת Fun ו- Call

פונקציית הרצה - run

```
(: run : String -> Number)
;; evaluate a FLANG program contained in a string
(define (run str)
  (let ([result (eval (parse str))])
    (cases result
      [(Num n) n]
      [else (error 'run "evaluation returned a non-number: ~s" result)])))
```

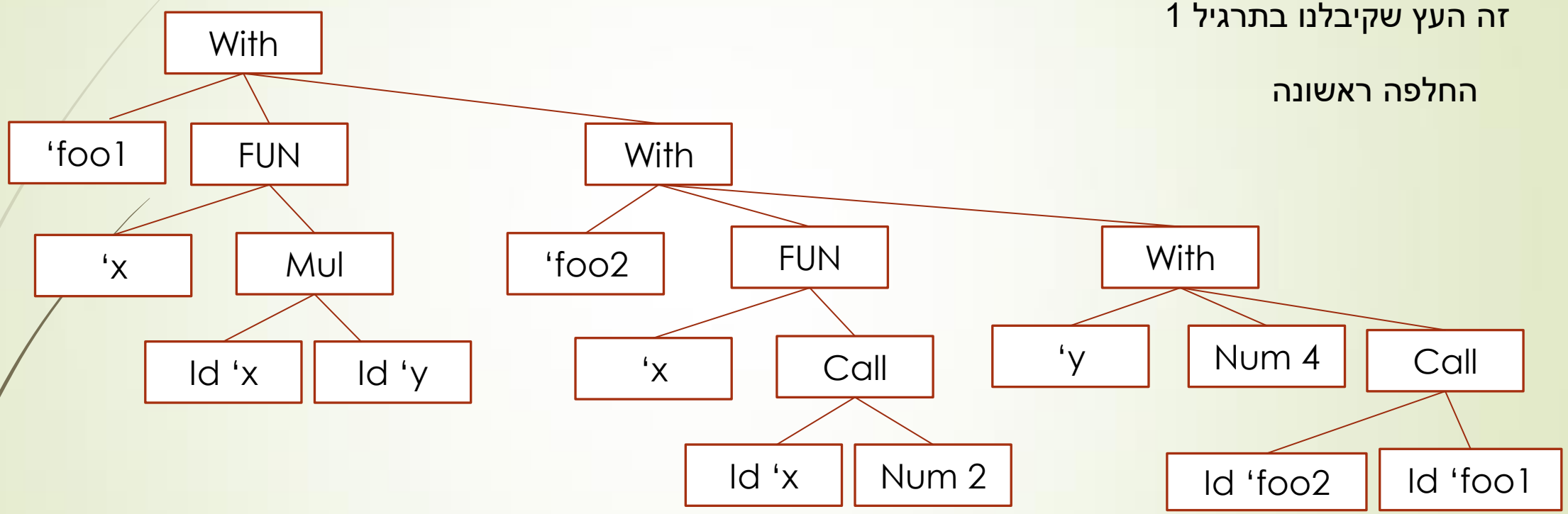
תרגיל 2

- בהמשך לתרגיל 1 - בתהליך ההערכה (הפונקציה eval) של ביטוי שקיבלנו יתבצעו חמש פעולות החלפה (הפונקציה subst) – בפעולה כזו מוחלף קדקוד בעץ (שנוצר עם בנאי Id) בתת-עץ אחר. ציירו את העץ המתקבל לאחר כל פעולת החלפה כזו (סה"כ ציירו חמישה עצים לפי סדר הופעתם בחישוב).

Ast1

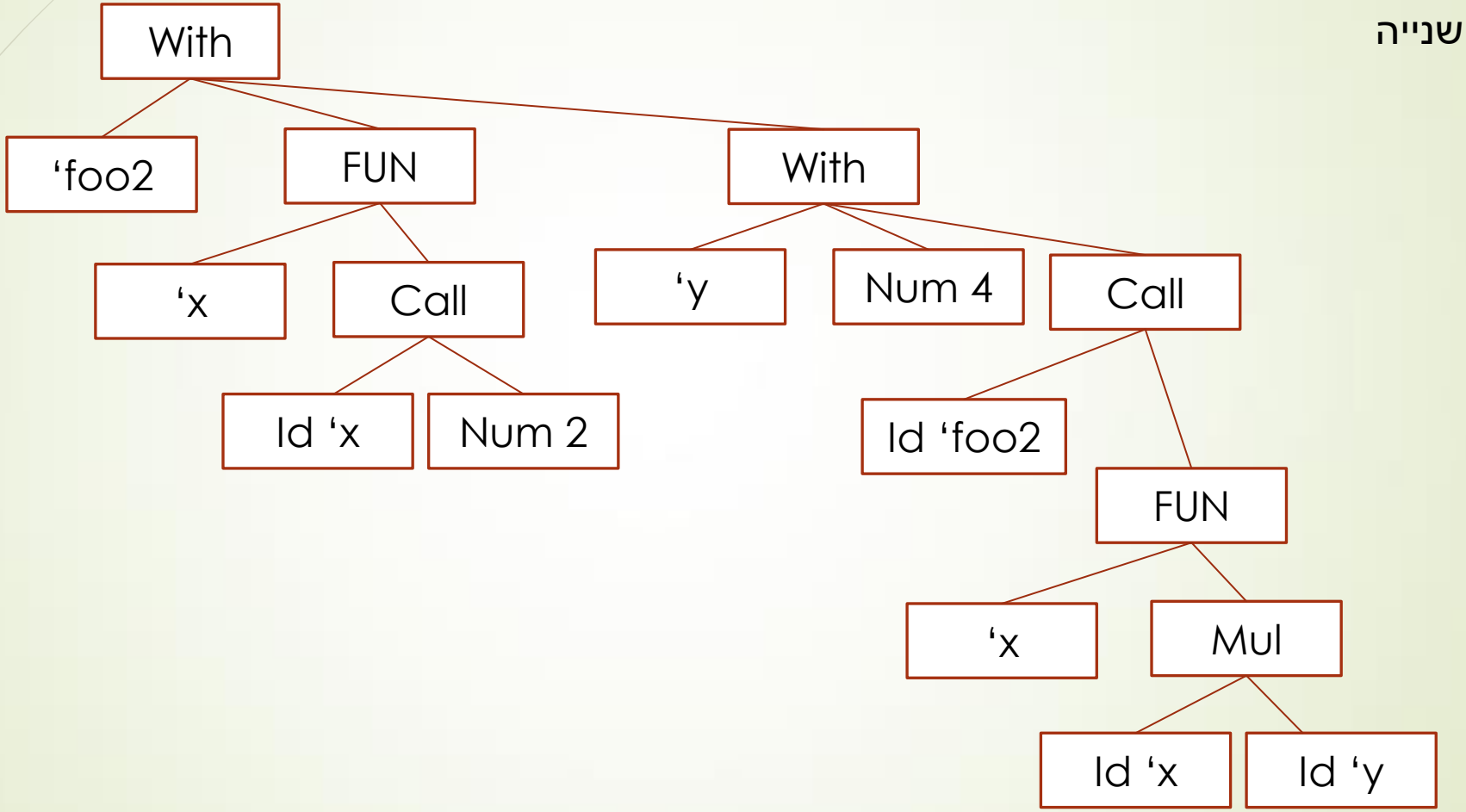
זה העץ שקיבלנו בתרגיל 1

החלפה ראשונה

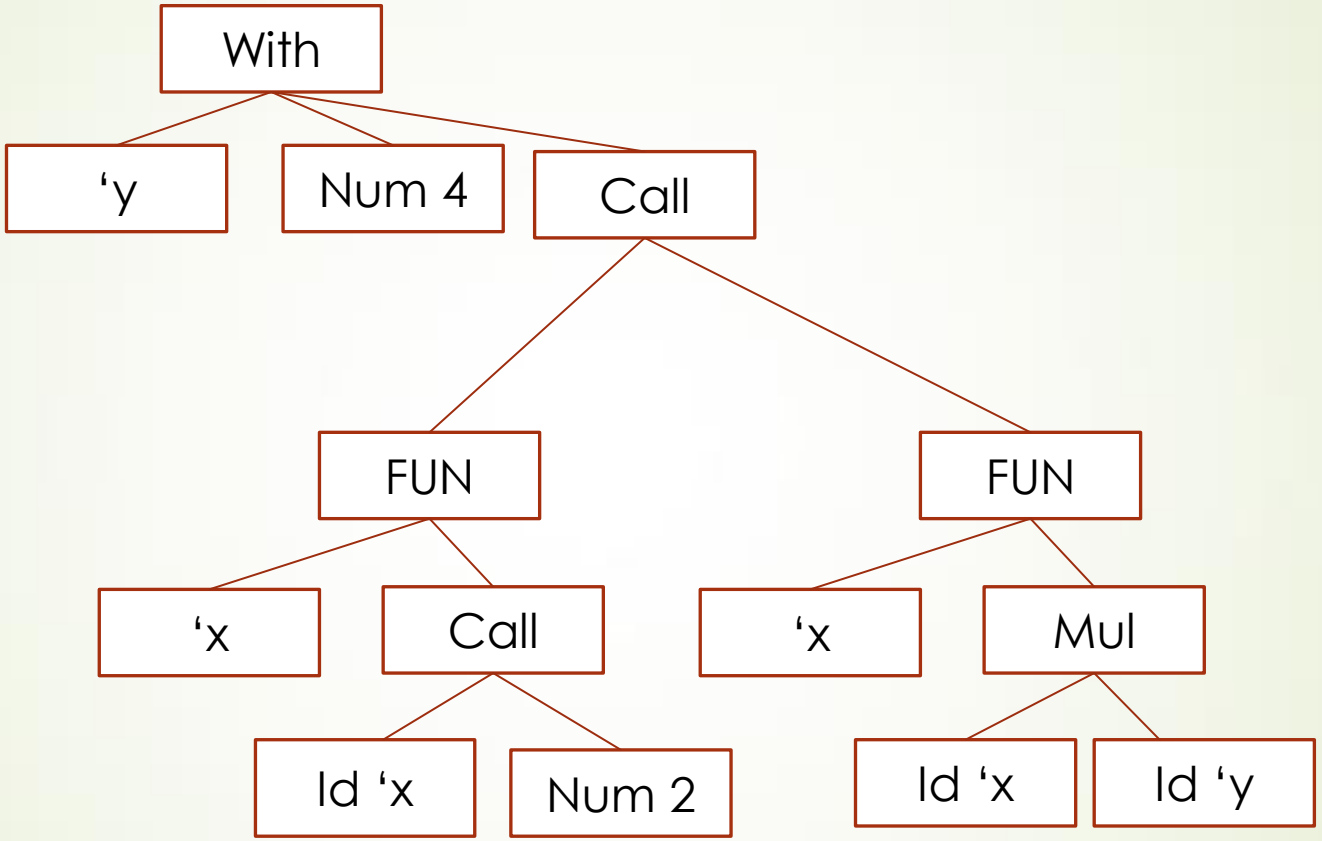


Ast2

החלפה שנייה

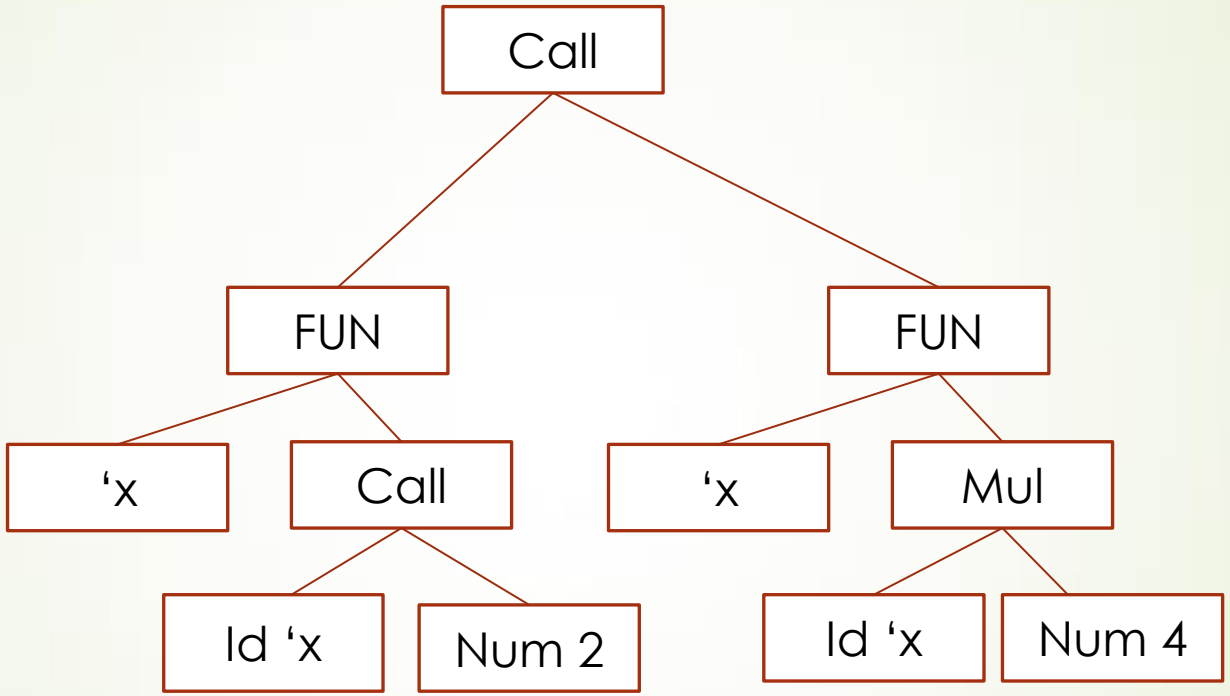


Ast3



החלפה שלשית

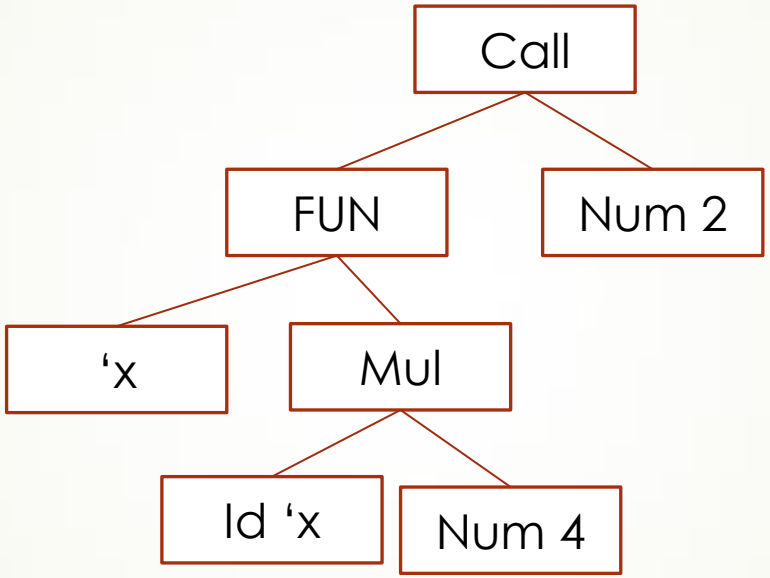
Ast4



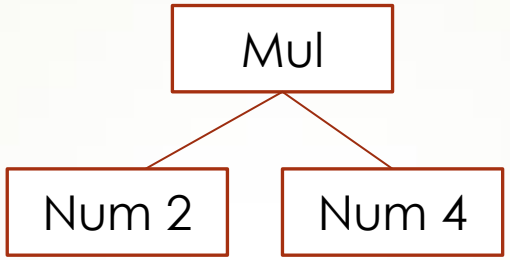
החלפה רביעית

Ast5

החלפה חמישית



Ast6



Ast7

Let vs. lambda

בRacket - Let הוא syntactic sugar עבור קריאה ל-lambda (פונקציה אנונימית).

אם נסתכל על הקוד הבא:

```
#lang racket
(let ([x (+ 3 2)]) (* x x))
```

זה יהיה שקול ל:

```
#lang racket
((lambda (x) (* x x)) (+ 3 2))
```

בשניהם אנחנו מקשרים את x לערך מסוים, במקרה הזה ל-(2 3 +), ומבצעים את פעולת החישוב הרצויה, במקרה הזה (x x *).

נשים לב שניתן להקביל את With ו-Fun & Call ל-Let ו-lambda. כלומר, With הוא syntactic sugar עבור Fun & Call.

תרגיל 3

- השתמשו ב- Call וב-Fun בשביל להגדיר את פעולת with.

כלומר, הגדירו מחדש את ה- parser על with, בעזרת הבנאים של Fun ו-Call.

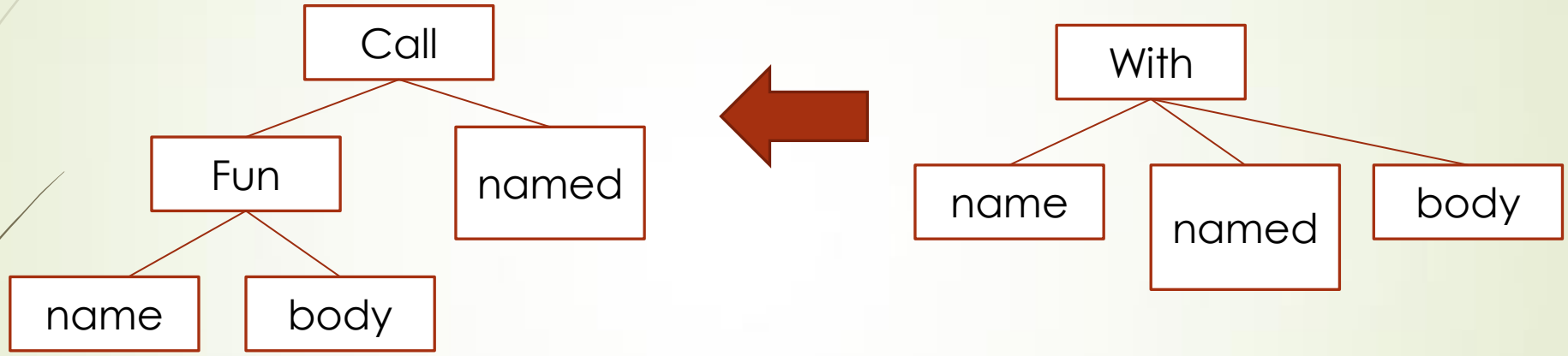
Old parser

```
(: parse-sexpr : Sexpr -> FLANG)

 (define (parse-sexpr sexpr)
   (match sexpr
     [(number: n)    (Num n)]
     [(symbol: name) (Id name)]
     [(cons 'with more)
      (match sexpr
        [(list 'with (list (symbol: name) named) body)
         (With name (parse-sexpr named) (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
     [(cons 'fun more)
      (match sexpr
        [(list 'fun (list (symbol: name)) body)
         (Fun name (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
     [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
     [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

New parser

```
(: parse-sexpr : Sexpr -> FLANG)

 (define (parse-sexpr sexpr)
   (match sexpr
     [(number: n)    (Num n)]
     [(symbol: name) (Id name)]
     [(cons 'with more)
      (match sexpr
        [(list 'with (list (symbol: name) named) body)
         (Call (Fun name (parse-sexpr body) ) (parse-sexpr named) )]
        [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
     [(cons 'fun more)
      (match sexpr
        [(list 'fun (list (symbol: name)) body)
         (Fun name (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
     [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
     [(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]
     [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

המחשה:



תרגיל 4

אילו פעולות צריך לשנות ב eval או בsubst בשביל להתאים לשינוי שעשינו?

תשובה:

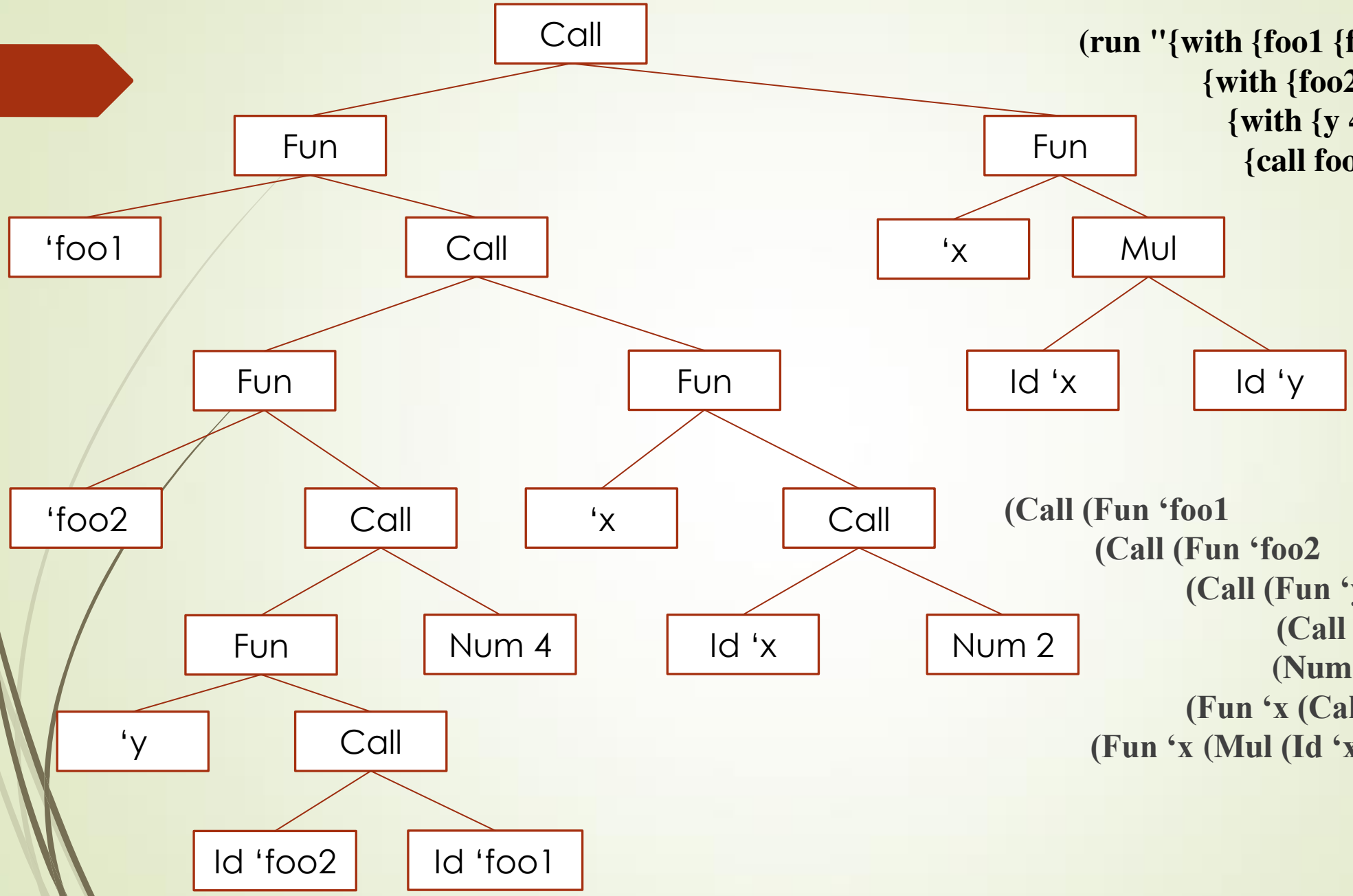
כלום, ניתן להסיר את הבנאי With מכל התת פרצדורה של הeval. אך לא חובה!

תרגיל 5

- ציירו את עץ התחביר האבסטרקטי המתאר את הביטוי הנתון במרכאות (כלומר את התוצאה של הפעלת parse על ביטוי זה).

```
(run "{with {foo1 {fun {x} {* x y}}}
        {with {foo2 {fun {x} {call x 2}}}
          {with {y 4}
            {call foo2 foo1}}}}")
```

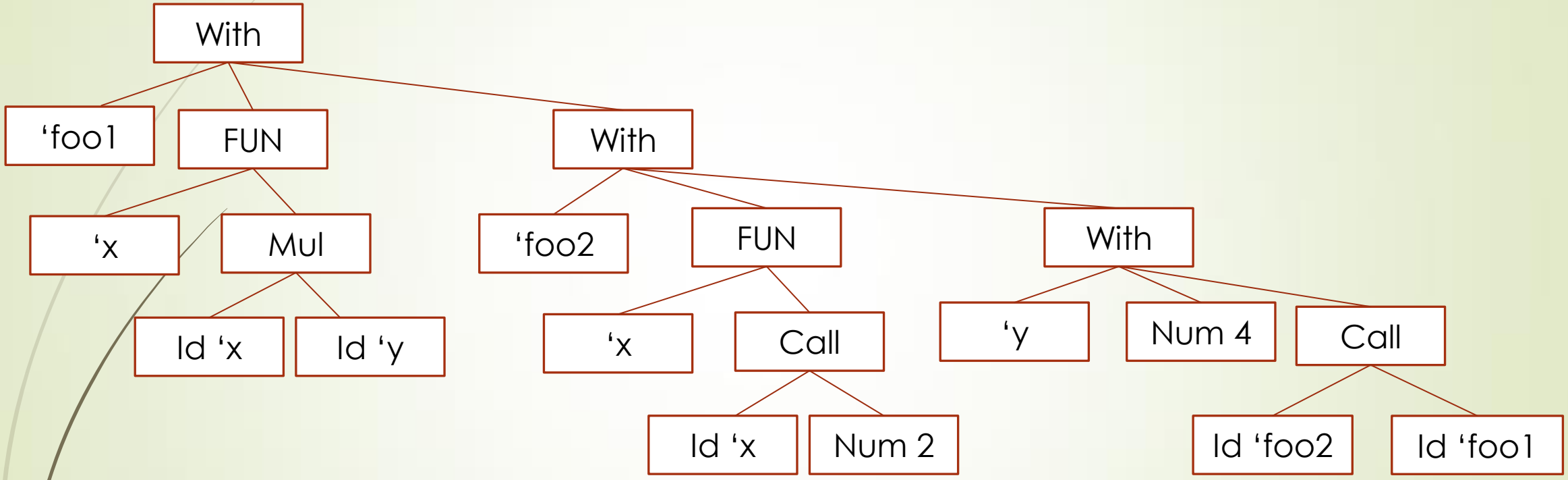
```
(Call (Fun ‘foo1
        (Call (Fun ‘foo2
                (Call (Fun ‘y
                        (Call (Id ‘foo2) (Id ‘foo1)))
                      (Num 4)))
              (Fun ‘x (Call (Id ‘x) (Num 2)))))
      (Fun ‘x (Mul (Id ‘x) (Id ‘y))))
```




```
(run "{with {foo1 {fun {x} {* x y}}}
        {with {foo2 {fun {x} {call x 2}}}
          {with {y 4}
            {call foo2 foo1}}}}")
```

```
(Call (Fun 'foo1
        (Call (Fun 'foo2
                (Call (Fun 'y
                        (Call (Id 'foo2) (Id 'foo1)))
                      (Num 4)))
                (Fun 'x (Call (Id 'x) (Num 2)))))
        (Fun 'x (Mul (Id 'x) (Id 'y))))
```

```
(run "{with {foo1 {fun {x} {* x y}}}
        {with {foo2 {fun {x} {call x 2}}}
          {with {y 4}
            {call foo2 foo1}}}}")
```

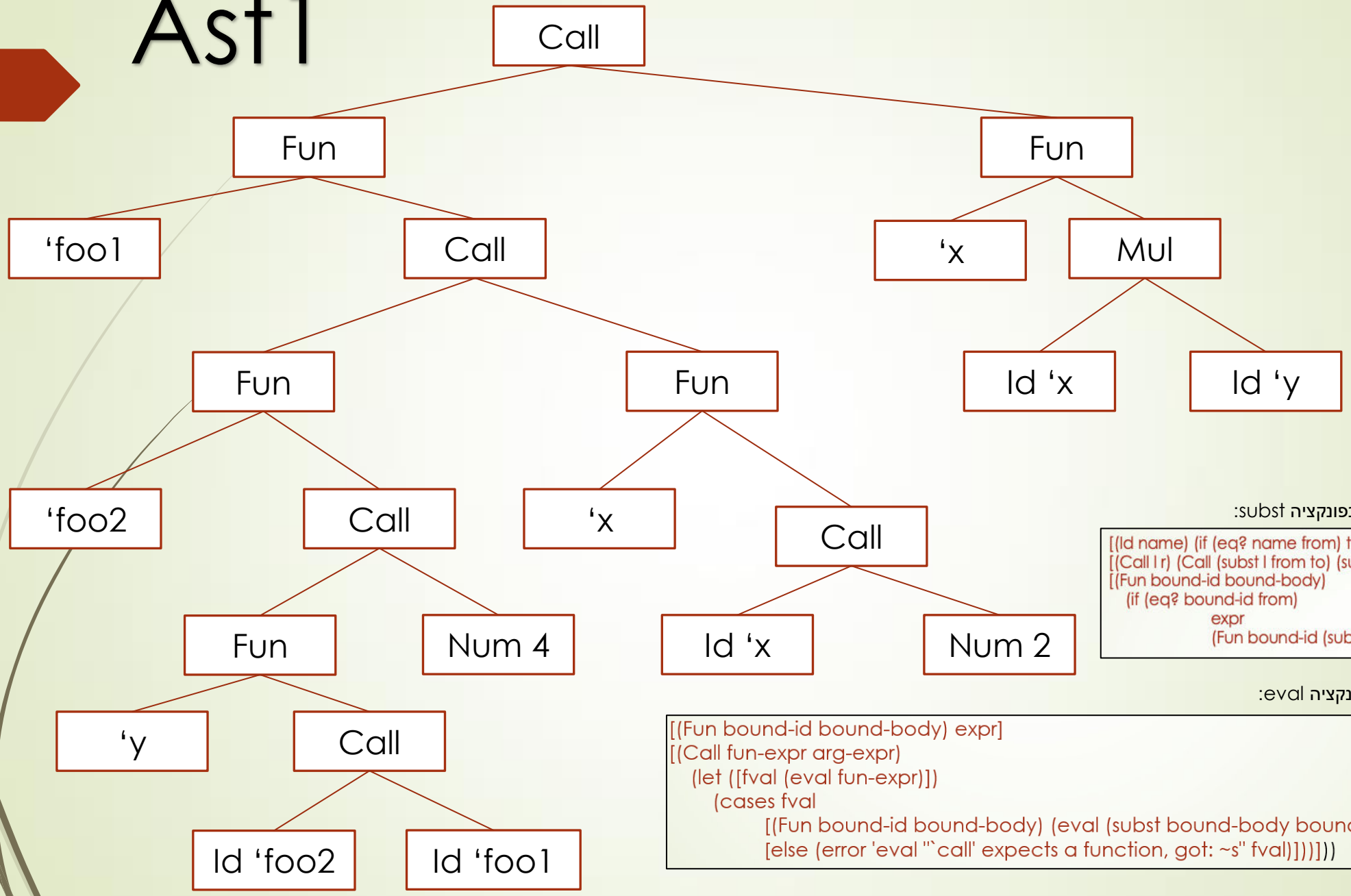


תרגיל 6

- בהמשך לתרגיל 5 – העריכו את הביטוי שהתקבל.

נזכיר שבתהליך ההערכה (הפונקציה eval) של ביטוי יתבצעו פעולות החלפה (הפונקציה subst) – בפעולה כזו מוחלף קדקוד בעץ (שנוצר עם בנאי ld) בתת-עץ אחר. ציירו את העץ המתקבל לאחר כל פעולת החלפה כזו

Ast1



החלק המתאים בפונקציה subst:

```
[(Id name) (if (eq? name from) to expr)]
[(Call l r) (Call (subst l from to) (subst r from to))]
[(Fun bound-id bound-body)
   (if (eq? bound-id from)
            expr
            (Fun bound-id (subst bound-body from to)))]))
```

החלק המתאים בפונקציה eval:

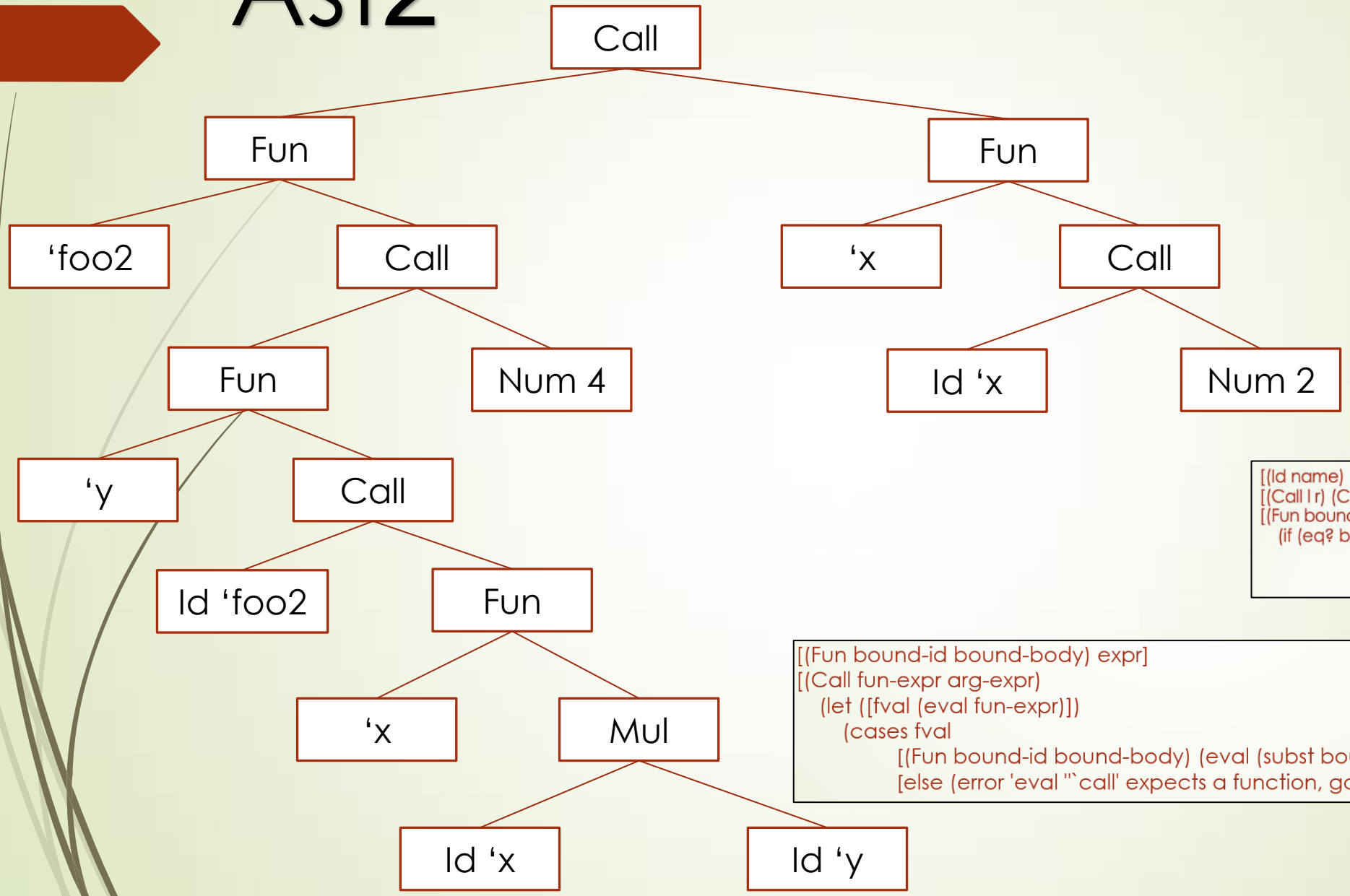
```
[(Fun bound-id bound-body) expr]
[(Call fun-expr arg-expr)
   (let ([fval (eval fun-expr)])
      (cases fval
            [(Fun bound-id bound-body) (eval (subst bound-body bound-id (eval arg-expr)))]
            [else (error 'eval "`call' expects a function, got: ~s" fval)]))]))
```

נשים לב לחלק החשוב בsubst!

```
(: subst : FLANG Symbol FLANG -> FLANG)
(define (subst expr from to)
  (cases expr
        [(Num n) expr]
        [(Add l r) (Add (subst l from to) (subst r from to))]
        [(Sub l r) (Sub (subst l from to) (subst r from to))]
        [(Mul l r) (Mul (subst l from to) (subst r from to))]
        [(Div l r) (Div (subst l from to) (subst r from to))]
        [(Id name) (if (eq? name from) to expr)]
        [(Call l r) (Call (subst l from to) (subst r from to))]
        [(Fun bound-id bound-body)
            (if (eq? bound-id from)
                expr
                (Fun bound-id (subst bound-body from to)))]))
```

בכל פעם שרואים בנאי Fun, בודקים את הדבר הבא:

Ast2



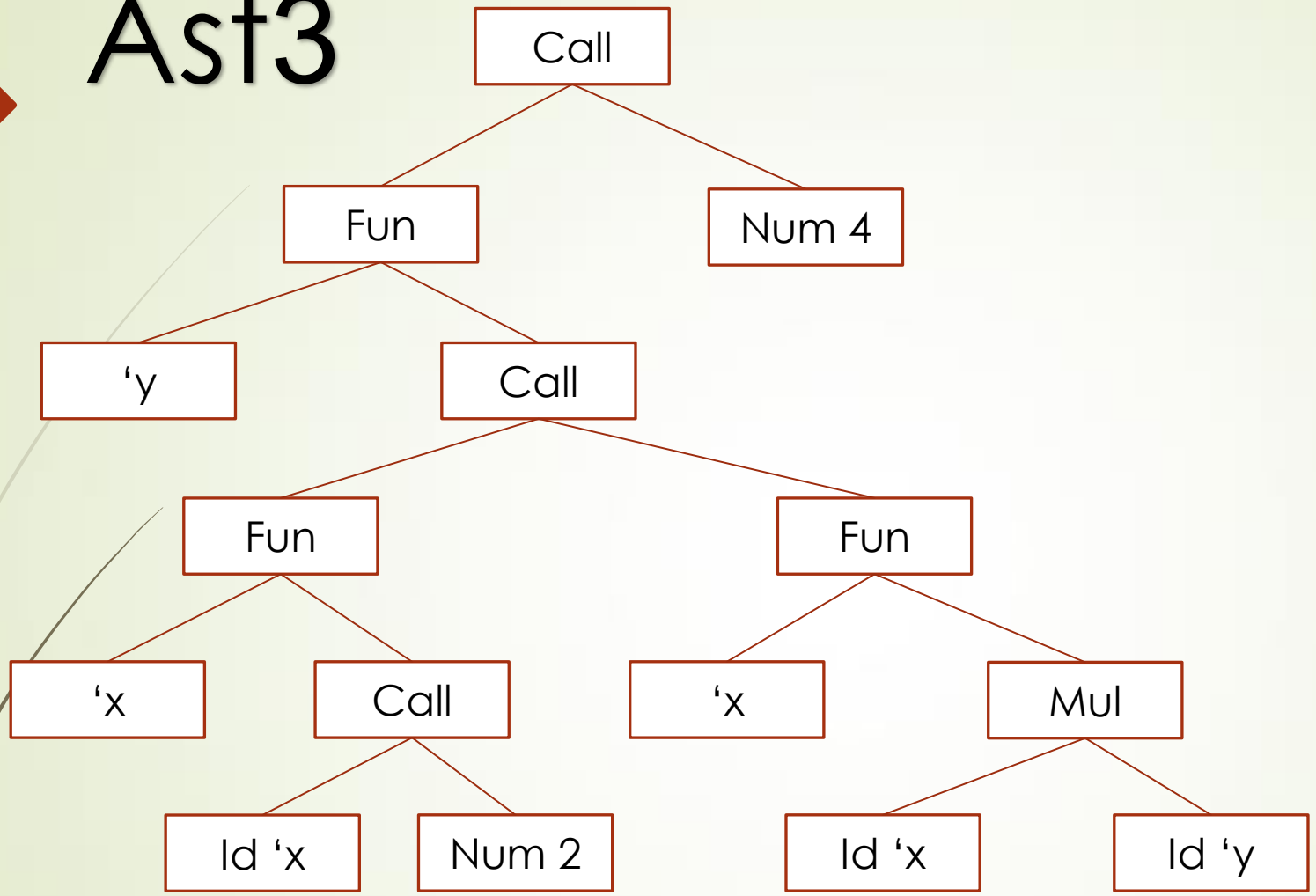
החלק המתאים בפונקציה subst:

```
[(Id name) (if (eq? name from) to expr)]
[(Call l r) (Call (subst l from to) (subst r from to))]
[(Fun bound-id bound-body)
   (if (eq? bound-id from)
            expr
            (Fun bound-id (subst bound-body from to)))]))
```

החלק המתאים בפונקציה eval:

```
[(Fun bound-id bound-body) expr]
[(Call fun-expr arg-expr)
   (let ([fval (eval fun-expr)])
      (cases fval
            [(Fun bound-id bound-body) (eval (subst bound-body bound-id (eval arg-expr)))]
            [else (error 'eval "`call' expects a function, got: ~s" fval)]))]))
```

Ast3



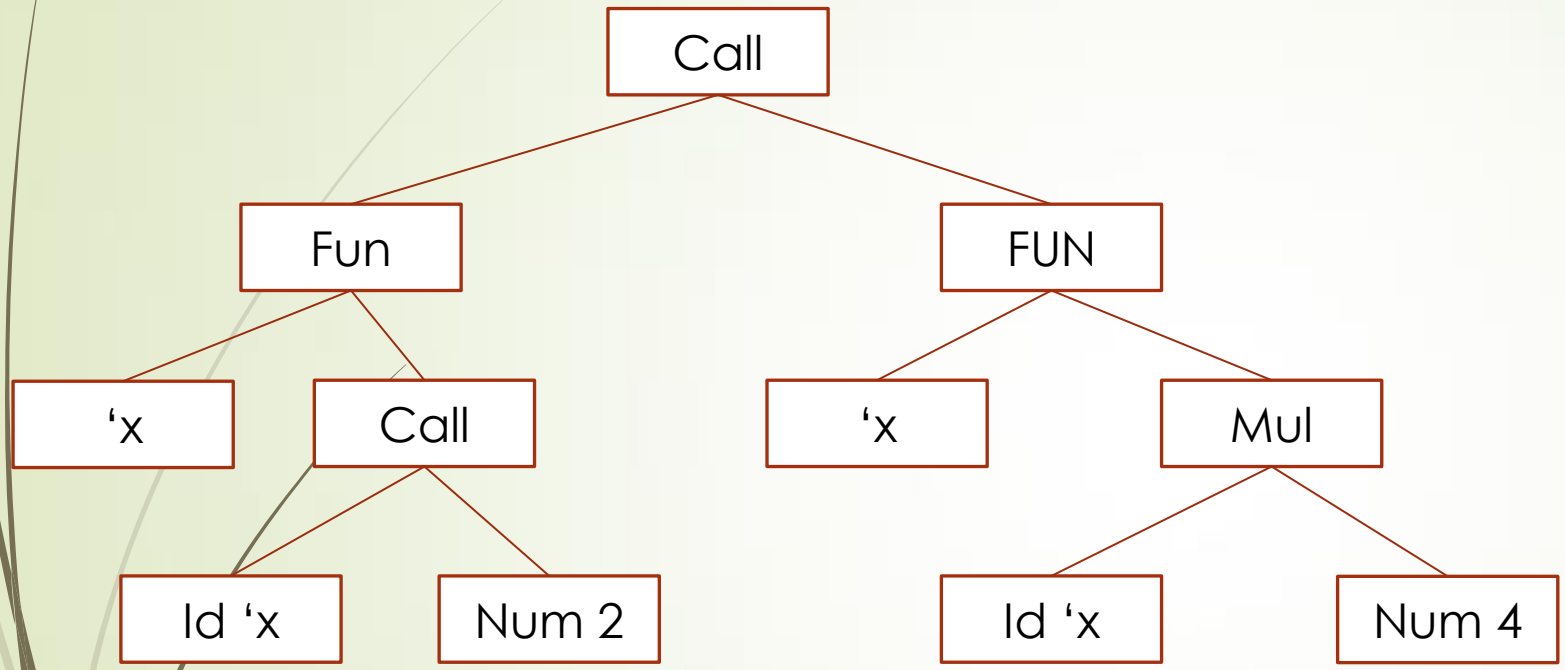
החלק המתאים בפונקציה subst:

```
[(Id name) (if (eq? name from) to expr)]
[(Call l r) (Call (subst l from to) (subst r from to))]
[(Fun bound-id bound-body)
   (if (eq? bound-id from)
            expr
            (Fun bound-id (subst bound-body from to)))]))
```

החלק המתאים בפונקציה eval:

```
[(Fun bound-id bound-body) expr]
[(Call fun-expr arg-expr)
   (let ([fval (eval fun-expr)])
      (cases fval
            [(Fun bound-id bound-body) (eval (subst bound-body bound-id (eval arg-expr)))]
            [else (error 'eval "`call' expects a function, got: ~s" fval)]))]))
```

Ast4



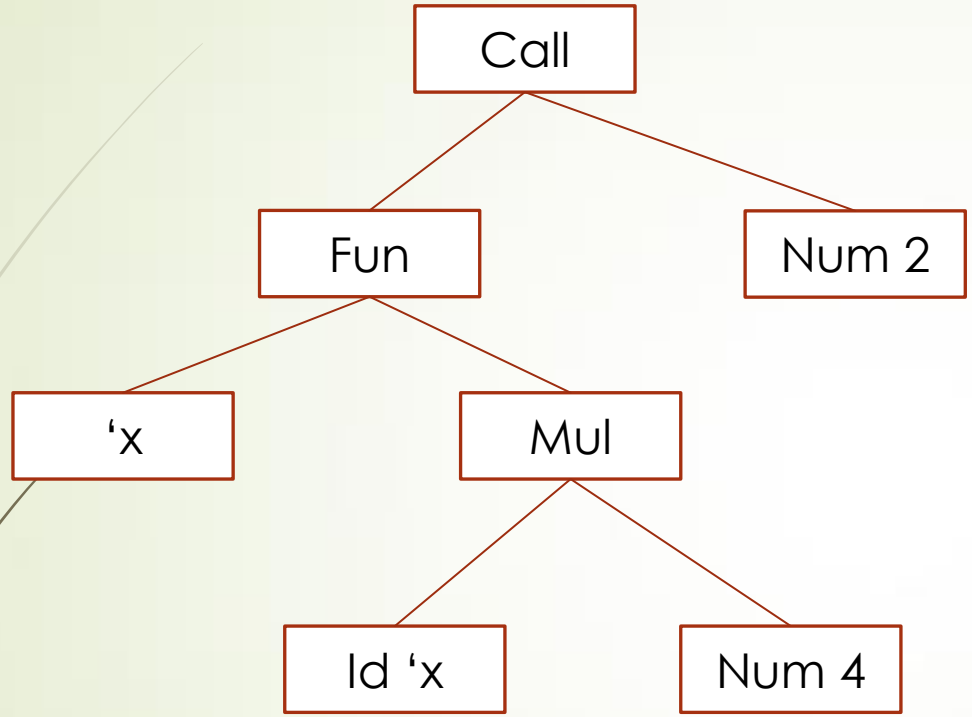
החלק המתאים בפונקציה subst:

```
[(Id name) (if (eq? name from) to expr)]
[(Call l r) (Call (subst l from to) (subst r from to))]
[(Fun bound-id bound-body)
   (if (eq? bound-id from)
            expr
            (Fun bound-id (subst bound-body from to)))]))
```

החלק המתאים בפונקציה eval:

```
[(Fun bound-id bound-body) expr]
[(Call fun-expr arg-expr)
   (let ([fval (eval fun-expr)])
      (cases fval
            [(Fun bound-id bound-body) (eval (subst bound-body bound-id (eval arg-expr)))]
            [else (error 'eval "`call' expects a function, got: ~s" fval)]))]))
```

Ast5



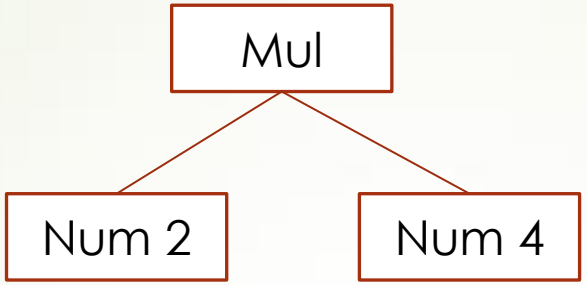
החלק המתאים בפונקציה subst:

```
[(Id name) (if (eq? name from) to expr)]
[(Call l r) (Call (subst l from to) (subst r from to))]
[(Fun bound-id bound-body)
   (if (eq? bound-id from)
            expr
            (Fun bound-id (subst bound-body from to)))]))
```

החלק המתאים בפונקציה eval:

```
[(Fun bound-id bound-body) expr]
[(Call fun-expr arg-expr)
   (let ([fval (eval fun-expr)])
      (cases fval
            [(Fun bound-id bound-body) (eval (subst bound-body bound-id (eval arg-expr)))]
            [else (error 'eval "`call' expects a function, got: ~s" fval)]))]))
```

Ast6



החלק המתאים בפונקציה eval:

```
[(Add l r) (arith-op + (eval l) (eval r))]
[(Sub l r) (arith-op - (eval l) (eval r))]
[(Mul l r) (arith-op * (eval l) (eval r))]
[(Div l r) (arith-op / (eval l) (eval r))]
```

הפונקציה arith-op:

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; `Num' wrapper
(define (arith-op op expr1 expr2)
        (: Num->number : FLANG -> Number)
        (define (Num->number e)
                (cases e
                        [(Num n) n]
                        [else (error 'arith-op "expects a number, got: ~s" e)]))
(Num (op (Num->number expr1) (Num->number expr2))))
```

Ast7