מרצה: ד"ר עומרי ערן
שעות קבלה: יום ג' או ה' (לתאם), חדר 11.2.11
מייל: omrier@ariel.ac.il

הקורס מבוסס על מה שהובא בעמוד הראשון במצגת (קישורים להרחבות).

נעבור לדבר על שפת התכנות של הקורס - השפה: Racket וסביבת העבודה DrRacket. (בסופו של דבר נעבוד עם שפת pl שהיא תת שפה של Racket).

כל שפה מורכבת משני דברים: סינטקס (תחביר = כללים, שרק אז הקומפיילר יקבל זאת) וסמנטיקה (מה משמעות הדבר - למה הקומפיילר יתרגם את הדברים).

בעמוד 3 ניתן לראות דוגמא - במערך הוגדרו רק 26 מקומות, ובכל השפות לא ניתן לדבר על המקום ה-30! בסטנדרט של כל השפות הוגדר מה יקרה (לרוב - אקספשיין), מלבד C - שם לא הוגדר מה יקרה, והכל תלוי בקומפיילר. ומדוע? כי רצו לתת כוח למתכנת, אך לאחר שעברו כמה שנים - הבינו שהרעיון הזה אינו טוב, וצריך לשמור שהמתכנת לא יחרוג.

אם כן, הדבר החשוב ביותר - הסמנטיקה! איך נלמד על הסמנטיקה? אפשר להשתמש בכלים פורמליים, אבל אנחנו נלמד בעזרת זה שנתכנן שפה ונבנה את הסמנטיקה שלה ומתוך כך נלמד את הסמנטיקה של כלל השפות.

נעבור להרצאה של ד"ר מגרמניה, שגם הוא מלמד את השפה של אלי.

תחילה יש להגדיר את השפה בא אנחנו משתמשים:

```
#lang pl
```

יש ברקט אלמנטים אטומים -

- true, false (ללא קשר לאפס ואחד) באותו מידה אפשר לכתוב #t, #f.
- מספרים - יש יצוג רגיל - מספר טבעי, יש יצוג למספר רציונאלי, גם $1/2$ זה יצוג (של חצי).
- מחרוזות - כמו שאנחנו רגילים בשאר השפות.
- סימבולים - אוסף תווים, אך אין זה מחרוזת (לא מעניין אותנו איך זה קיים שם), לפני הסימבול מוסיפים תג, למשל:

```
'sym
```

ההבדל בין סימבול למחרוזת: סימבול לא ניתן להפריד או לשרשר, לעומת מחרוזת שכן ניתן. בסימבול לא ניתן לשים רווח או עוד גרש, אך כל השאר אפשרי.

- תווים - #\a זהו התו a. ניתן לקחת כמה תווים ולהפוך אותם למחרוזת, ובאותה מידה לקחת מהמחרוזת תו בודד. דוגמא יוצאת דופן:

```
#\space      ;#\ רווח בודד
```

הקוד ברקט הוא פריפיקס - לכן האופרטור יהיה ראשון (ויש לשים את המשפט בסוגריים עגולים).
<u>שרשור מחרוזות</u>:

```
(string-append "a" "b") - שרשור מחרוזות, אפשר גם יותר מ-2
(string-ref "apple" 0) - החזרת התו במקום מסויים, למשל: עבור המיקום אפס
```

<u>הפונקציה eq</u> - לא בודק ערך, אלא אותו אובייקט ממש, ולכן בדיקה של eq על פרמיטיבי - זה יחזיר אמת. לעומת השוואה של 2 מחרוזות - לא בטוח שהן באותו מיקום.

```
(eq? 'x 'x) - האם האובייקטים זהים
(eq? "ap" "ap") -> true
```

לעומת equal ששואל על התוכן. גם string=? יחזיר את אותה התשובה, אלא שהיתרון הוא - פונקציה המיועדת לטיפוס מסוים.

```
(equal? "ap" "ap") ->true
```

null - רשימה ריקה! ולכן null? זוהי שאלה - האם מדובר ברשימה ריקה. באותה מידה ניתן לשאול האם מדובר במחרוזות, האם מדובר במספר וכו'?, למשל:

```
(string? "abc") -> true
(number? 123) -> true
```

הערות - מספיק נקודה פסיק, אך על שורה שלימה ד"ר ערן דורש לרשום 2 נקודה פסיק. ניתן להשתמש בבלוק שלם: #|.

תנאי if - נכתוב if באופן הבא: מה לעשות אם התנאי מתקיים, ולאחר מכן - מה לעשות אם התנאי אינו מתקיים, למשל:

```
(if (< 2 1) 'a 'b) -> 'b     בגלל ש-2 לא קטן מ-1
```

ברקט - לכל ביטוי יש ערך מוחזר, ולכן חייבים בתנאי של ה:if לעשות "else".

יש אופציה נוספת ליצור תנאי- בעזרת המילה השמורה cond. היתרון הוא - שיותר קל ליצור יותר מתנאי אחד, למשל:

```
(cond
    [(< 2 1) 17]
    [( > 2 1) 18])  -> מחזיר 18
```

סוגריים מרובעים זהים לעגולים, אך זו מוסכמה בשבילנו שנדע שזהו תנאי פנימי. ה:cond עובד שלב אחרי שלב, ו:else צריך להיות אחרון. אם משהו התקיים - שם זה נעצר. למען הסדר הטוב - דורש ד"ר ערן שבכל cond יהיה רשום else, ובדוגמא לעיל:

```
(cond
    [(< 2 1) 17]
    [else 18])
```

חשוב לשים לב - גם else דורש סוגריים מרובעות.

במקרה שלא נרשום else - ייתכן ושום תנאי לא יתקיים ולכן יוחזר ערך (void) - וזה לא תקין לפי דרישותיו של ד"ר ערן (לא מפיל את התוכנה).

הערך מה-cond חוזר, ויש בעצם מישהו שמחכה לתשובה מה:cond שרץ.

דבר נוסף: ניתן ליצור תנאי שבוודאות יתקיים, כמו למשל:

```
(cond
    […]
    [true 8])
```

ז"א: התנאי האחרון - בוודאי יתקיים, אם אף תנאי קודם - לא התקיים. באותה מידה - ניתן לכתוב תנאי שלא יכול בכלל לרוץ - בתנאי שלעולם הוא לא יתקיים

```
(cond
    […]
    [true 8]
    [false (* 'a 'b)])
```

המשך המצגת בשיעור הבא.

נמשיך לעבור על המצגת (מושג 13) עם מושג ה"רשימה". יוצרים רשימה בעזרת המילה list, ואם מדובר בזוג סדור נשתמש במילה cons. חשוב לשים לב - בשביל ליצור זוג - הצמד השני חייב להיות null, ז"א:

```
(cons 1 null) - > '((1))
(cons 1 (cons 2 null))
```

אפשר גם להשתמש ב: ( ) ' בשביל ליצור רשימה, למשל:

```
(list 1 2 3)
'(1 2 3)
```

צורת הרשימה בזיכרון בנויה באופן הבא: עבור הרשימה:

```
'(1 2 3 ())
```

אנחנו בעצם מדברים על זוגות סדורים - הערך הראשון - הינו 1, והערך השני - שאר איברי הרשימה. אם נכנס לאיבר השני - גם שם נמצא זוג סדור, שהראשון הוא 2 והשני שאר איברי הרשימה, וכו' וכו'.
בסופו של דבר יש לנו רשימה עם 4 איברים - 1,2,3 ורשימה ריקה.

פונקציות שניתן להפעיל על רשימות:

שרשור של רשימות: append, למשל:

```
(append (list 1 2) '(3 4))
```

חשוב לשים לב - שרשור רשימה לרשימה ריקה - יחזיר את הרשימה הלא ריקה - כמו שהיא. אם ברצוננו לשרשר רשימה ריקה - יש לשרשר רשימה ובתוכה איבר null, מה שאומר - רשימה ובתוכה רשימה ריקה, למשל:

```
(append (list 1 2) (list null)) -> '(1 2 ())
```

- פונקציה להחזרת האיבר הראשון ברשימה - first, למשל:

```
(first (list 1 2 3)) -> 1
```

חשוב לשים לב - שמה שמוחזר - הינו הערך ברשימה, ובדוגמא לעיל - integer.

- פונקציה להחזרת הרשימה ללא האיבר הראשון: rest, למשל:

```
(rest (list 1 2 3)) -> '(2 3)
```

כאן - מה שמוחזר זו רשימה. ובמקרה שיש רק איבר אחד ברשימה - יוחזר null, ז"א: רשימה ריקה.

- פונקציה להחזרת איבר לפי מיקום - list-ref, למשל:

```
(list-ref (list 1 2 3 4) 3) -> 4
```

הגדרת משתנים ופונקציות - [עמוד 15 במצגת]

ניתן להגדיר משתנה בודד בעזרת המילה השמורה define, למשל: (define PI 3.14)
דוגמא להגדרת פונקציה קבועה:

```
(define (double x)
     (list x x))
```

בעצם הגדרנו פונקציה בשם double שמקבלת משתנה x (מכל סוג שהוא), ומחזירה רשימה של x x.

כאשר נרצה להגדיר פונקציה שמקבלת משתנים ספציפיים - נגדיר תחילה את הפונקציה - מה היא מקבלת ומה היא מחזירה, למשל: פונקציה הבודקת את גודל הרשימה:

```
(: length : (Listof Any) -> Natural)
(define (length l)
     (cond
          [(null? l) 0]
          [else (add1 (length (rest l)))]))
```

הערה: (Listof Any) - הינה רשימה המכילה איברים מכל סוג שנרצה.

הגדרת define-type - הגדרת משתנה חדש. לכל משתנה - נוכל ליצור כמה "בנאים" שנרצה, למשל:

```
(define-type Animal
     [ Snake Symbol Number Symbol ]
     [ Tiger Symbol Number ])
```

לאחר הגדרת ה:define-type מסוג animal- נוכל להשתמש בבנאים שירצנו, למשל:

```
( Snake 'Slimey 10 'rats )
( Tiger 'Tony 12)
```

כאשר אני יוצר אובייקט חדש - racket עוטף אותי בפונקציות אוטומטיות. למשל: אפשר לבדוק האם מדובר באובייקט מסוג ?animal ללא שכתבנו את הפונקציה הזו בעצמנו.
דוגמא נוספת: הפונקציה cases המקבלת וריאנט של הטיפוס, היא בודקת באיזה בנאי מדובר, ובעצם יש כאן תנאי - מה לעשות "אם"... דוגמא:

```
(cases (Snake 'Slimey 10 'rats)
     [(Snake n w f ) n ]
     [(Tiger n sc ) n ])
```

הפונקציה בודקת מהו הבנאי - ומוצאת את הבנאי - ומקשרת את השדות, ולכן מוחזר שמו של בעל החיים.

חשוב לשים לב: ניתן להשתמש באיחוד המסומן ב:U (יו גדול), מה שמאפשר ליצור משתנה חדש שהוא איחוד של שני אחרים (אפ' יותר מ-2), לדוגמא: פונקציה שתוכל להחזיר או מספר או ערך בוליאני:

```
(: animal-weight : Animal - > (U Number #f))
( define (animal-weight a)
     (cases a
          [(Snake n w f) w]
          [else #f]))
```

<u>תנאים לוגים</u> - ניתן להכניס יותר משני משתנים עבור and או or, דוגמא פשוטה:

```
(and true true)
```

חשוב לשים לב: אם בדקנו תנאי של and, ואחד התנאים לא התקיים - רקט לא ימשיך לבדוק את שאר התנאים. כנ"ל לגבי or, רק הפוך - אם מצאנו תנאי שמחזיר ture - לא נבדוק את המשך התנאים.

דוגמא נוספת - (and true 1) - אם מתקיים true - מחזיר את 1.

let - הגדרה של טיפוס לוקלי. ז"א: אחרי החישוב שנעשה בהם - הם "נעלמים". בזמן הגדרת המשתנים - המחשב עדיין לא יודע על קיומם, ולכן בדוגמא:

```
(let ([x 0])
    (let ([x 10]
          [y (+ x 1)])
          (+ x y)))
```

ה-x הפנימי שווה 10, ולאחר החישוב החיצוני התוצאה תהיה שווה ל-11.

*let - זה let מקונן, ואז התוצאה תהיה 21 (לא יהיה בשימוש בקורס).

אפשר להגדיר <u>פונ' אנונימית</u>, ז"א: הגדרת פונקציה ללא שם. משתמשים במילה השמורה lambda. אך אנחנו פחות נתעסק עם זה, כנ"ל לגבי הנושאים: 28-29.

שים לב: ישנו כלי חזק בשם map - שמקבל פונקציה וליסט, ומפעיל את הפונקציה על כל איבר בליסט. בנוסף: ישנו כלי בשם andmap - המקבל פונקציה בולאנית - ובודק האם כל איברי הליסט עומדים בתנאי של הפונ'. (כנ"ל לגבי ormap).

<u>הערות מהתרגול</u>: ניתן להפעיל פונקציות בצורה רקורסיבית, כפי שאנו רגילים. אך צריך להכיר גם את המושג - רקורסיית זנב, ז"א: הפונקציה המקורית - מפעילה פונקציית עזר, המקבלת משתנה נוסף שזוכר את החישוב לאורך כל הדרך.

נעבור לדבר על הגדרת שפה (מצגת מס' 3). אנו רוצים להגדיר שפת תכנות. נתחיל עם השפה הפשוטה ביותר:

```
<AE> ::= <num>
       | <AE> + <AE>
       | <AE> - <AE>
```

באופן כללי, כאשר אנו נדבר על <num> בעצם נדבר על כל המספרים, אפשר לראות שיש שני כללים נוספים, ובעצם לא נייצר מילה עד שכל המילה תהיה מורכבת מכלל מס' 1 (מספרים בלבד).

כאשר אפשר לקרוא את אותה מילה בשני אופציות ויותר - מדובר ב:BUG (נקרא Ambiguity) והדקדוק הנ"ל באמת לא נכון... ז"א: אם אפשר לייצר את אותה מילה משני עצים - זה לא טוב. ולכן אנו זקוקים לפונקציה - שלכל מילה יש דרך יצור אחת בלבד.

אפשר לראות שבדוגמא <NUM> <NUM> כבר יש בעיה בייצוג של 3 2 1.

היום ננסה למצוא פתרון לבעיה מהשיעור הקודם (המשך מצגת 3). במהלך השיעור הראה ד"ר ערן כמה פתרונות אפשריים - אך בכל אחד גילנו בעיה, בסופו של דבר הפתרון הנכון - הוספת סוגריים, מה שיובא לקמן (הדרך לפתרון הנכון - לא סוכמה בצורה מיטבית).

תחילה חשבנו לפתור את הבעיה באופן הבא: אפשר לומר שתמיד השמאלי יהיה DIGIT (ראה במצגת עמוד 3), ז"א: אות לא סופית, ובעצם אני מחייב שממנה תצא רגל בודדת. בהמשך המצגת (עמ' 4) ניתן לראות מקרה של <AE> + <AE> שהוא כבר עם הבעיה הנ"ל. בנוסף: אם נרצה סדר קדימויות - הדבר הראשון שצריך לקרות - צריך להיות למטה:

```
<AE>     :: <num>
         |  <AE> + <AE>
         |  <FAC>

<FAC>    ::  <num>
         |   <FAC> * <FAC>
```

בעצם -מכפל לא ניתן לחזור לחיבור.

כמובן שבעיית ה:Ambiguity רק גדלה באופן הזה, כי אפ' לספרה בודדת יש 2 דרכים - ישיר מ:AE, או מ:AE ל:FAC ומשם למספר הבודד. ובעצם ניתן לוותר על ה:num הראשון.

בשביל לכפול סכומים - נצטרך להשתמש בסוגריים, ראה עמוד 5, ופתרון אחרון בעמ' 6.

אפשר לראות שבסופו של דבר - אם יש סוגריים - אנחנו פותרים את הדו-משמעות, ז"א: יש לנו עץ בנוי באופן קבוע, וכך אנחנו יכולים לדבר על שפת תכנות. ולכן הפתרון הפשוט ביותר:

```
<AE>     ::  <num>
         |   ( <AE> + <AE> )
         |   ( <AE> - <AE> )
```

למשל:

$$\left(\left((1-2)+3\right)+(4-5)\right)$$

אפשר לשים לב שאת הכלל הראשי אנחנו מחלקים בדיוק ל-2 כאשר ה:+ באמצע.

זה הבסיס לשפה שלנו, אך נעשה מעט שינוי (בשביל לא להתבלבל עם רקט) - הסוגריים שלנו יהיו מסולסלות. אך כן ניקח את המוסכמה של רקט - האופרטור משמאל לביטוי, למשל: { + <AE> <AE> }, ז"א: prefix.

(מצגת מס' 4) אם נרצה להמיר את העץ בשפה שאין בה Ambiguity - נסתכל על הפעולה כפונקציה, מה שייתן לנו את העץ בצורה רקורסיבית.

בעצם נצטרך לממש ברקט את מבנה הנתונים שמייצג את הסינטקס האבסטרקטי - ז"א: את העץ. בעצם נגדיר טיפוס חדש, והכל בעצם יקרה מעצמו.

```
(define-type AE
     [Num Number]
     [Add AE AE] - טיפוסים רקורסיבים
     [Sub AE AE]
     [Mul AE AE]
     [Div AE AE])
```

אם כן, בכל פעולה אנחנו מפעילים בנאי, כמו למשל: Add. וכל מספר בודד הוא בעצם עלה.

```
((3-4)+7)   ->   { + { - { 3 4 } } 7 }
```

זוהי התוכנית הפשוטה בשפה שלנו.

בעצם עכשיו יש צורך לעבור את תהליך הפרסינג, ז"א: המתכנת שלנו יכתוב את התוכנה שלו (כמו הדוגמא לעיל), ובעצם היא תתקבל כמחרוזת לפרסר (מוסכמה ביננו לבין המתכנת). כעת הפרסר יקח את המחרוזת ויבנה עץ סינטקס אבסטרקטי, ז"א: AST. הדרך לעשות את זה - לעבור דרך מבנה נתונים שיותר קל לעבוד איתו, ושמו: Sexpr, שזה: או סימבול, או מספר או רשימה. ז"א: יש כאן שני שלבים - ממחרוזת ל:Sexpr ומשם ל:AST. החלק הראשון כבר נעשה בשבילנו (בשפה Pl), משתמשים בפונקציה: string->sexpr, למשל:

```
(string->sexpr "3") -> 3
(string->sexpr "{+ 2 3}") -> '(+ 2 3)
```

חשוב לשים לב: המחרוזת "7" פשוט תהפוך ל:7. אך אם מדובר בביטוי - הוא יעבור ל:list, למשל:

```
'(+ ( - 3 4 ) 7 )
```

הגרש - נכנס רקורסיבית, ובעצם - +, - שניהם סימבולים.

במהלך השיעור הראה ד"ר ערן את התהליך בשביל להגיע ל: parse-sexpr, אך בסיכום הבאנו את המסקנה בלבד: בעמ' 3 ניתן לראות שאפשר לוותר על make-node בעזרת match, ואפילו לבנות את כל הפונקציה בעזרתה. יש לה מילים שמורות - למשל: number: בודקת אם מדובר ב:Number, ואם כן - תקשר אותו ל:n וכו'. ראה בסיכום להלן - את הפונקציה parse-sexpr).

לסיכום עד כה:

```
(define-type AE   ; הגדרה של משתנה חדש
    [Num Number] ; רשימת הבנאים
    [Add AE AE]
    [Sub AE AE])

(: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n) (Num n)]
      [(list '+ left right)
       (Add (parse-sexpr left) (parse-sexpr right))]
      [(list '- left right)
       (Sub (parse-sexpr left) (parse-sexpr right))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> AE) ; מפעילים את הפונקציה שלנו על הפונ' של אלי
  ;; parses a string containing an AE expression to an AE
  (define (parse str)
    (parse-sexpr (string->sexpr str)))
```

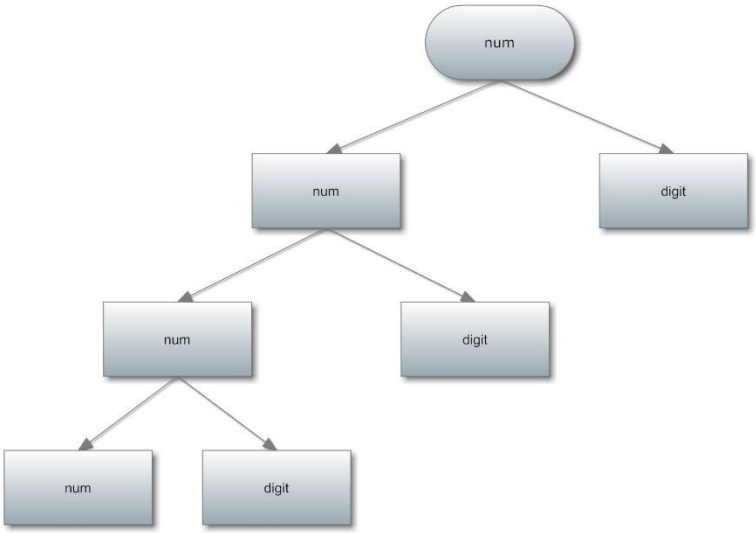
דוגמא להרצה:

```
(parse "{+ 2 3}") -> (Add (Num 2) (Num 3))
(parse-sexpr '(+ 2 3)) -> (Add (Num 2) (Num 3))
```

בסוף השיעור הביא ד"ר ערן הסברים מפורטים עבור הפונקציה match (בפועל לא עברנו עליהם בשיעור, אך מומלץ לחזור על כך, החל מעמוד 4 במצגת מס' 4).

נמשיך בנושא משבוע שעבר, רק הפעם נרצה לתת ל"חיבור" את הערך של חיבור וכן הלאה. בעצם נעבור למושג של eval, ז"א: אחרי שיש בידנו עץ ast - איך אני נותן משמעות לעץ, ובמקרה שלנו - חיבור וחיסור. נושא זה מופיע במצגת מס' 5.

אנו נחייב לבנות את העץ בצורה קבועה כך שלא יהיה Ambiguity. בעצם מדובר בתכונה של הדקדוק, ואנו שואלים האם קיימת פונקציית eval שמחשבת את תת העץ הנוכחי (הכל יחושב בצורה רקורסיבית).



במהלך השיעור הראה ד"ר ערן קוד שאינו נכון (הצגה של מספר תלת ספרתי). בעצם אני רוצה לראות אם אני יכול לחשב את הערך של ה:eval של כל העץ...
בסופו של דבר, התיקון שעשינו הוא בהגדרה של השפה (= הבנאים). כך ש:

```
eval(digit) = digit ; נשאר אותו דבר
eval(Num, digit) = 10*eval(num) + eval(digit)
```

לכן, לדוגמא, עבור 103 - נקבל 103 ולא תשובה אחרת.

כעת נבנה eval לשפה שלנו - שיקבל AE ויחזיר מספר רגיל (פעולות אריטמטיות על מספרים).

parse על 3 אמור להחזיר cons num על 3. וכן לגבי:

$$\left(parse(+\ 3\ 4)\right) \rightarrow Add$$

כאשר add עם שני בנים - num 3, num 4.

תזכורת: AE זו הגדרה שלנו לטיפוס חדש עם שלושה בנאים, זה בעצם type חדש עם פונקציות ממשק בחינם, כגון: cases. ניתן לשים לב שאם קיבלתי בנאי של add היא בעצם מקבלת שני AE.

לאחר כל ההקדמות הגענו אל הקוד עצמו, חשוב לשים לב - ה:bnf רק בשבילנו כי הוא בהערה בכלל:

```
#| BNF for the AE language:
     <AE> ::= <num>
            | { + <AE> <AE> }
            | { - <AE> <AE> }
            | { * <AE> <AE> }
            | { / <AE> <AE> }
|#
```

בסופו של דבר הקוד שלנו יראה כך:

```
(: eval : AE -> Number)
  (define (eval expr)
    (cases expr
      [(Num n)   n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]))
```

ניתן להוסיף פונקציה נוספת שמפעילה את כל התהליך שעשינו עד כה, ז"א: מקבלת מחרוזת - הופכת אותה ל:sexpr, ומ:sexpr לעץ ast, ומשם לחישוב הערך - בעזרת eval, ז"א:

```
(: run : String -> Number)
  ;; evaluate an AE program contained in a string
  (define (run str)
    (eval (parse str)))
```

נעבור למצגת מס' 6: נרצה כעת לייעל את השפה, למשל: אם יש קוד שחוזר על עצמו - נרצה לתת לו שם ולהשתמש בו (ובמילים גבוהות: השמה הינה binding, והחלפת המשתנה במילים גבוהות הינה substitution), ואפי' יותר מזה - מחשב אותו פעם אחת ונותן לו שם. מה היתרון? יעילות, ואף יותר מזה - לתת משמעות ספציפית לחישוב. בנוסף: מניעת חזרות - לא צריך לחזור ולחפש בקוד את החישוב ולתקנו.

תחילה נגדיר את with ללא משמעות, ולאחר מכן נוסיף לו משמעות.

נרצה בעצם להגדיר משתנה בשם x למשל, ולומר שהוא (+ 4 2). נשתמש לשם כך בסינטקס של אלי:

```
{with {x {+ 4 2}} {* x x}}
```

x יישאר חי לאורך כל הבלוק, לאחר מכן הוא יישכח...

כמובן שאפשר להגדיר גם with בתוך with. (אין להגדיר 2 with במכה אחת). בעצם אנחנו מגדירים מעין let חלש יותר.

כעת נרחיב את השפה, נוסיף with לשפה שלנו, תחילה נגדיר זאת לעצמו ב:bnf:

```
 | { with { <id> <WAE> } <WAE> }
```

בעצם נרחיב גם את ה:define-type, ונקבל תוכנית חדשה: נשתמש ב:W גדולה ב:With עבור בנאי.

```
(define-type WAE
     [Num Number]
     [Add WAE WAE]
     …
     [With Symbol WAE WAE])
```

כאשר:

א. Symbol - זהו המשתנה שיקבל את הערך.
ב. WAE הראשון - זהו הערך שישלח אל המשתנה Symbol
ג. WAE השני - זה ה:Body (כל המשך הקוד- גוף ה:with).

חשוב לשים לב שחסר לנו בנאי נוסף - לדעת לדבר על סימבולים (לא רק להכריז), ולכן נוסיף בנאי בשם id:

```
[id Symbol]
```

חשוב שלא לשכוח להוסיף גם ב:bnf, ז"א:

```
<WAE> ::= <num>
          ...
          | { with { <id> <WAE> } <WAE> }
          | <id>
```

כעת יש להוסיף את with לפונקציה של ה-parser: (חשוב לשים לב, במהלך ההרצאה - נתן ד"ר ערן את הפתרון בשבלים, ושילב אופציות שקורסות במקום להחזיר הודעת שגיאה - התוספת מסומנת בצהוב).

```
(: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)     (Num n)]
      [(symbol: name)  (Id name)]
      [(cons 'with more)
       ;; go in here for all sexpr that begin with a 'with
       (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)]))
```

תחילת השיעור עשה ד"ר ערן חזרה על סוף השיעור האחרון. דוגמא לקוד עם with:

```
{with {x 7}
     {+ x 5}}
```

חשוב לזכור: ל-with - יהיו תמיד שלושה בנים (אפשר לראות זאת על פי ההגדרה, וכן בדוגמא לעיל). במקרה שלנו בניו הם x - symbol, Num 7 והשלישי: Add, שגם לו 2 בנים: 'x id, Num 5.

עדיין לא הגדרנו את הסמנטיקה של with, ואת זה נעשה היום - eval על with.
חשוב לשים לב - ה-x הראשון הוא הצהרה, והשני - הוא השימוש בו.(כל זה בעמ' 3 במצגת).

כעת נרצה לתת משמעות ל-with, ז"א: מה יקרה כאשר המתכנת יכתוב את המילה with. ברור שאני רוצה שתהיה כאן החלפה. ז"א: הכנסת ערך לנעלם שבתוך ה-body. הסימון לכך הוא:

```
WAE2[WAE1/id]
```

ז"א: קח את 2 - תחליף בו את id תמורת הערך 1.

על פי ההגדרה הראשונית שהבאנו בשיעור - אם יהיה with בתוך with עם אותו symbol - נקבל שגיאה, למשל:

```
{with {x 5} {+ x {with {x 3} 10}}}  ->  {+ 5 {with {5 3} 10}}
```

כי נגיע למצב שמספר מחליף מספר, ולא symbol. כיצד נתקן את הביטוי? בעצם צריך להגדיר את האיקס הפנימי כתפקיד אחר - הראשון id, והשני רק הצהרה. לפני שנסביר כיצד הדברים עובדים - נקדים ונאמר כי לכל חלק בביטוי יש תפקיד:

1. Binding Instance - ההצהרה הראשונה על המשתנה (binding) - לעולם לא נרצה להחליף אותו, זה בעצם id של with, ולא ה-'x id שאנו משתמשים בו. לדוגמא:

```
{with {x 7} {+ x 2}}
```

בעצם ה-x הראשון הינו Binding Instance.

2. scope - זהו התחום, איפה להחליף - אילו מופעים של x להחליף. למשל:

```
{with {x 7} {+ x 2}}
```

3. bound instance - אותם מופעים של x החייבים להיות בתוך ה-scope של x. למשל:

```
{with {x 7} {+ x 2}}
```

4. free instance - מופע של משתנה שלא קשור לאף bound instance. זה כמובן באג בתוכנית. אלא שאם נסתכל רקורסיבית על התוכנית ייתכן ונראה free instance, ולא יהיה מדובר בבאג, כפי שיוסבר לקמן.

עד כאן אין פתרון לשאלה, אבל הבנו את הבעיה - לא הפרדנו בין binding לבן bound.
ננסה לפתור את הבעיה - תחליף את כל המופעים של x בתנאי שאינו הצהרה.

אך גם זה לא נכון מכיוון אחר, כי לא נרצה להחליף מופע ב-scope אחר. למשל:

```
{with {x 5}
    {+ x {with {x 3}
           x}}}
```

בעצם, לפי ההגדרה הנ"ל - נקבל 10, במקום 8, כי נחליף גם את ה-x האחרון ב-5.

לכן נגדיר כך: עבור כל with - נחליף את x רק עבור אותו scope.

גם כאן תהיה בעיה, כי אנחנו "זהירים מדי", למשל:

```
{with {x 5} {+ x {with {y 3} x}}}
```

בעצם - כן הינו צריכים להחליף את המופע האחרון של x, ולא עשינו זאת...
כי with לא צריך לבצע החלפה - אלא רק אם מדובר באותו שם משתנה (כי אם למשל היה מדובר ב-y, ולאחר מכן x - היה באג, כי אנו זקוקים להגדרה של x). כאן התוכנית כבר תעבוד. אך זו הגדרה מסובכת למה שכבר ידענו לומר עוד קודם:
נאמר הפוך - תחליף את כל המופעים החופשיים בתוך ה:bound instance, ובכל השאר - לא לגעת. בעצם נשלח את תת הביטוי ללא הראש - ונחליף את החופשיים במשתנה שניתן בהצהרה Binding Instance.

במהלך השיעור - הראה ד"ר ערן את הפונקציה subst, חשוב לשים לב: שבשלב הראשון ישנה בעיה, ניתן עליה את הדעת בשיעור הבא...

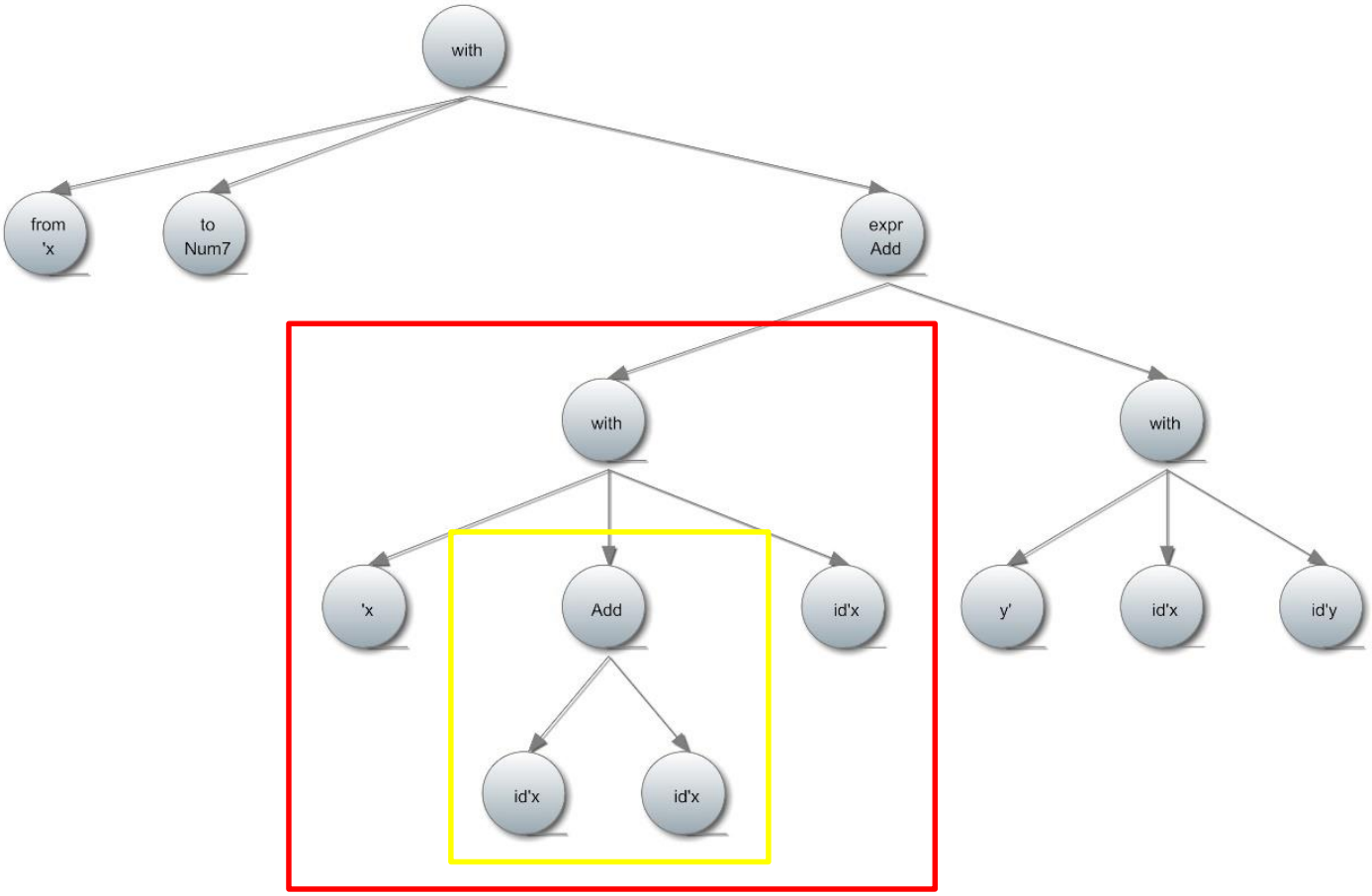
ברגע שאנחנו מפעילים את הפונקציה על הביטוי - אם מדובר ב:Num אין מה לעשות - פשוט להחזיר אותו, אחרת - אם יש פעולה - נחזיר רקורסיה על 2 החלקים - מה שמבטיח לי שאין משתנים חופשיים ב-2 החלקים (מהרישא כמובן).
אם קיבלנו id הוא בוודאי מוחלף - אם מדובר באותו משתנה, מדוע? כי נטפל בזה בתוך הבנאי של with - בעצם נימנע מלהיכנס לתתי עץ ששם המשתנה הוא זהה.

הקוד שהגענו אליו עד כה:

```
(: subst : WAE Symbol WAE -> WAE)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (define (subst expr from to) ; returns expr[to/from]
    (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)
       (if (eq? bound-id from)
         expr           ; <-- don't go in!
         (With bound-id
               named-expr
               (subst bound-body from to)))]))
```

בקוד הסופי שראינו עד כה - יש באג... שגם בו נצטרך לטפל, נעשה זאת בשיעור הבא (כנ"ל - לגבי המצגת, הפתרון מופיע בתחילת מצגת 7), שים לב - ישנם 2 באגים, נטפל בהם אחד אחרי השני.

להמשיך.
נחזור על הבעיה משבוע שעבר: נראה דוגמא של עץ:



הקוד של העץ הנ"ל הינו:

```
{with {x 7}
        {+ {with { x {x x x} x}
                {with {y x} y}}}
```

הבעיה - אנחנו כלל לא נכנסים ל-with השמאלי (מסומן באדום), וכאשר נרצה לחשב אותו - נהיה בבעיה, על אף שהקוד הוא חוקי, כי כאשר ננסה לעשות eval, עדיין לא חישבנו את הערך של {add id'x id'x} (מסומן בצהוב), וזה לא ניתן - כי יש לנו משתנה חופשי.

לכן הפתרון הוא - שלא להגיע לקודקוד בעייתי, ז"א: נפעיל בתוך ה:eval - subst שידאג שלא יהיו כלל משתנים חופשיים, ולכן אם נפעיל eval ונקבל id - זה אומר error.
מתוך כך - ב:subst נפעיל גם על ה:name-expr את subst, ובכל מקום שרשום id'x תחליף ל:Num7, ולאחר מכן - שנחשב את ה:with - נחליף את id'x ב-Num14, כי קודם נריץ subst.

בסופו של דבר התיקון בפונקציות יראה כך:

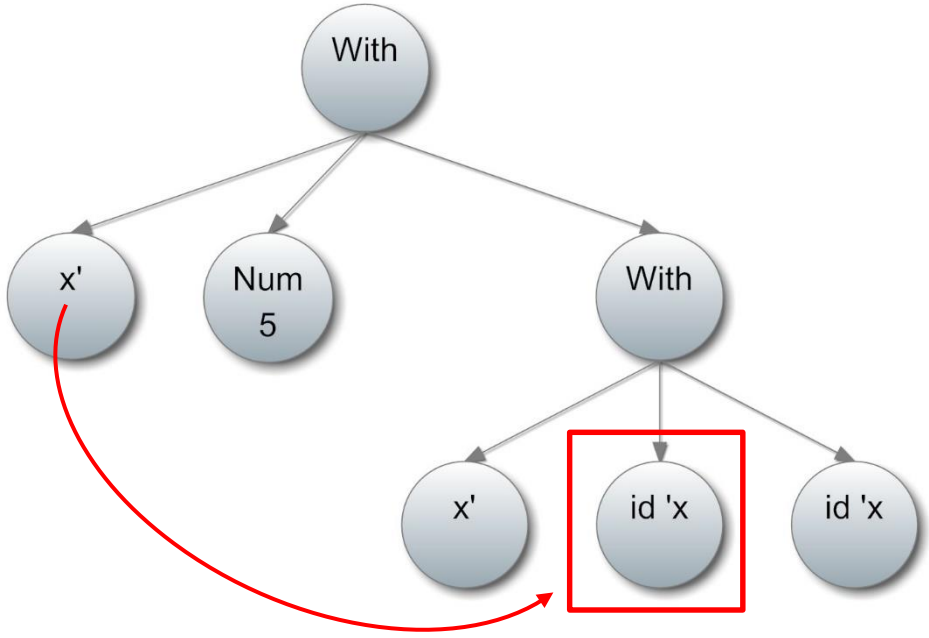
```
(: eval : WAE -> Number)
  ;; evaluates WAE expressions by reducing them to numbers
  (define (eval expr)
    (cases expr
      [(Num n) n]                                  ;; same as before
      [(Add l r) (+ (eval l) (eval r))]            ;; same as before
      [(Sub l r) (- (eval l) (eval r))]            ;; same as before
      [(Mul l r) (* (eval l) (eval r))]            ;; same as before
      [(Div l r) (/ (eval l) (eval r))]            ;; same as before
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
                    (Num (eval named-expr))))] ; <-*** (see note)
      [(Id name) (error 'eval "free identifier: ~s" name)]))

(: subst : WAE Symbol WAE -> WAE)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (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)
       (if (eq? bound-id from)
         expr
         (With bound-id
               (subst named-expr from to)        ; <-- new
               (subst bound-body from to)))]))
```

לאחר שתיקנו את הבעיה הראשונה - בכך שהפעלנו subst גם על named-expr, וכתבנו את eval היטב. כעת נשארה לנו בעיה נוספת (בעיה אחת לפני אחרונה ברשימת הבדיקות), ז"א:

```
(test (run "{with {x 5} {with {x x} x}}") => 5)
```

הבעיה בבדיקה הנ"ל - אנחנו כלל לא נכנסים ל:with השני - משום שהסימבול זהה - 'x.

מה שצריך לעשות - להכניס את ה:if פנימה, כי ייתכן ויהיו מקרים בהם נצטרך להיכנס, כמו בדוגמא משמאל, ולהחליף את ה:name-expr ב:'x הקודם (ז"א: - 5 בדוגמא לעיל - מסומן באדום)



```
(: subst : WAE Symbol WAE -> WAE)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (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) ;; new - only ask on body
               bound-body
               (subst bound-body from to)))]))
```

שיעור שעבר סגרנו את נושא ה:with. נעבור לנושא הפונקציות:

עד עכשיו ראינו את ההשמה של מספר אל x, כעת נרצה לתת את ההשמה בהמשך הדרך, למשל: לקבל את הערכים מהמשתמש, ונגדיר לעצמנו את fun - אם לא ניתן לה שם היא תהיה אנונימית.
בעצם fun - היא הרצה מיידית של with ללא ערך עבור הסימבול.

<u>הערה</u>: חשוב לשים לב: אם מדובר בפונקציה אנונימית - היא תחזיר פונקציה, למשל:
(lambda (x) (* x x))
אם נרצה שהיא תחזיר מספר כמו let נוסיף סוגריים עם השמה:
((lambda (x) (* x x)) 5)
ובעצם נקבל 25.

היתרון בפונקציה - שאין צורך לתת באופן מיידי את הערך. מה שייתן את האופציה - להימנע מכפל קוד.

נגדיר בשפה שלנו מילה עבור הפעלה של פונקציה: call.

איך נגדיר שם לפונקציה? בעזרת with, לדוגמא:

```
{with {sqr {fun {x} {* x x}}}
    {+ {call sqr 5}
       {call sqr 6}}}
```

בעצם אנחנו נתנו שם לפונקציה - sqr, ואז אנחנו מפעילים את הפונקציה על 5, ועל 6, ומחברים את התוצאה.

במהלך השיעור - דיבר ד"ר ערן על נושא הפונקציות באופן כללי - ברקט ובשאר שפות (עמ' 2-4), לא סוכם.

כעת נצטרך לשנות את השפה שלנו - תחילה נוסיף ל:bnf את הפונקציה, בשביל שנשים לב לשינוי - נשתמש בשפה בשם FLANG ונוסיף את מה שחסר בשביל לדבר על פונקציות:

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

נעבור לדבר על ה:dfine-type:

```
...
[Fun Symbol FLANG]
[Call FLANG FLANG]
```

אין חובה ש:fun וגם call יהיו ביחד (אם הם ביחד - זה דומה ל:with שעשינו).

נעבור לפרסר: כמעט הכל נשאר אותו דבר:

```
(: 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)])]
      [(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)]))
```

חשוב לשים לב: `(symbol: name)` אנחנו בודקים האם מדובר בליסט שהתוכן שלו הינו סימבול ונותנים לו שם: name. הסיבה לכך - משום שב: define-type הגדרנו את ה: symbol עם סוגריים.

נעבור למשמעות של fun - תחילה - אנחנו צרכים להבין איך לעשות את ההחלפה (substitution). עבור fun החוקיות של ה: scope זהה לזו של with, ועבור call - החוקיות של call זהה לשל הפעולות האריטמטיות.

```
(: subst : FLANG Symbol FLANG -> FLANG)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (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)))]))
```

כעת נדבר על המשמעות של הפונקציה - ז"א: eval. הוספנו טיפוס לעולם, כי עד עכשיו עסקנו רק במספרים, וכעת אנחנו עוסקים גם בפונקציות, ז"א: ייתכן ואני אקבל פונקציה ולא רק מספר. ואם אנחנו מקבלים

פונקציה - איך אנחנו "מחזיקים" אותה? ד"ר ערן הסביר על הלוח שאנחנו בעצם שומרים מעין "ענן" ששומר את החישוב עצמו. איך נשמור את ה"ענן" הזה? הוא בעצם קיים בתוך ה:FLANG, כי Fun בנוי כך:
(Fun 'x FLANG)
לכן אין צורך לבנות משהו חדש. ומכיוון שאני רוצה לשמור גם פונקציה וגם מספרים - נמיר את המספרים ל:Num (בעצם נעטוף אותם), ולכן תמיד נחזיר FLANG מ:EVAL (במקום מספר).

נעבור לחוקים של האבוליישן (להעתיק מהמצגת).

```
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
```

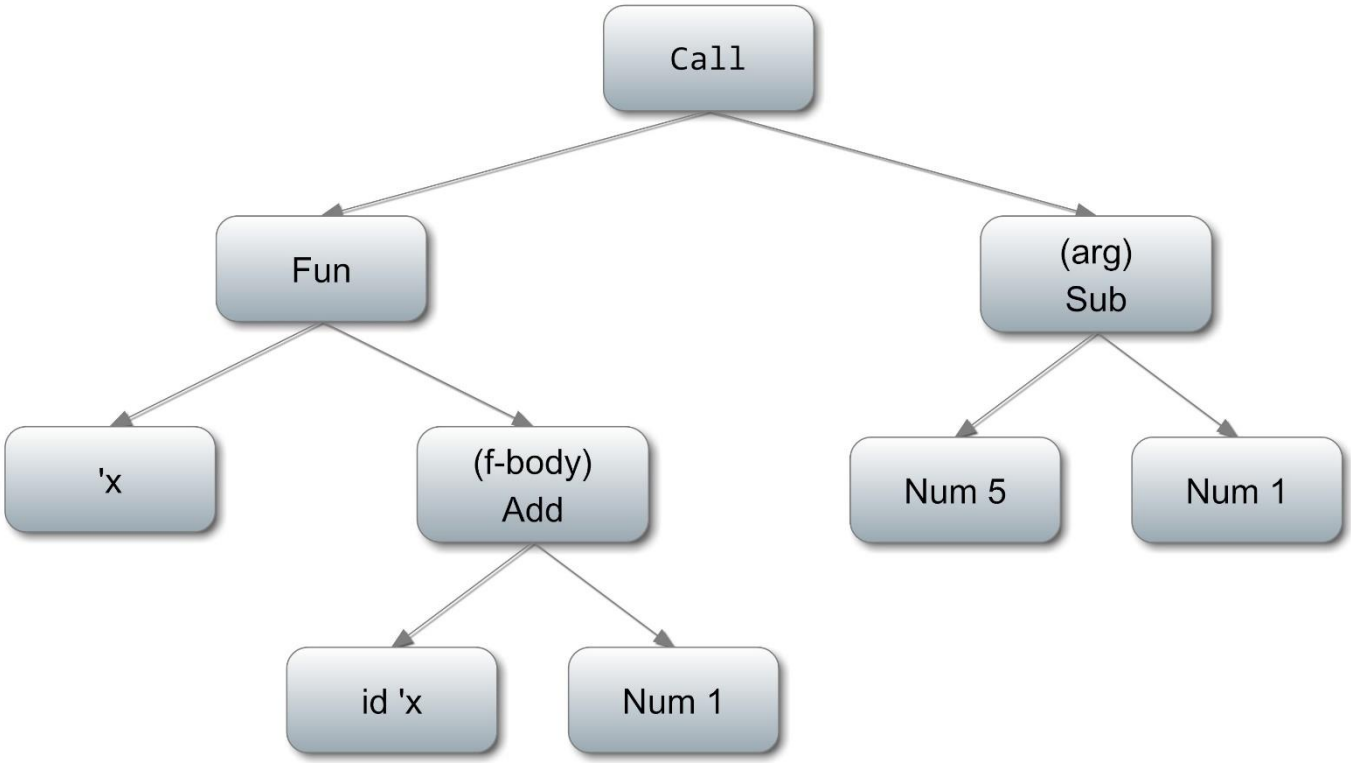
חשוב לשים לב שהחוקיות של call כמעט זהה ל:with, למעט שינוי הסדר. לקראת סוף השיעור הראה ד"ר ערן את הקוד עצמו, ואמר שיש בו באג, אך מכיוון שהדברים לא היו ברורים - נחזור עליהם בשיעור הבא.

נמשיך בנושא הפונקציות. נחזור על הדברים:

{call {fun {x} {+ x 1}} {- 5 1}}

אנחנו בעצם יוצרים פונקציה ומפעילים אותה על {- 5 1} השווה ל-4.

מה תוצאת ה:parse? נצייר עץ:



מהו ה:eval, נעשה זאת בשלבים

1. v = eval(arg-expr)
2. T = f-body[v/x]
3. eval(T)

דיברנו גם בשיעור שעבר - שמעכשיו ה:eval יחזיר flang, ולא מספר. ו:flang יהיה או Num או Fun.
חשוב לשים לב - החזרה של פונקציה לא יכולה להיות בתוספת של Num, ולכן add וכו' - יחזירו רק Num.
עוד יש לשים לב: eval של Fun מחזיר פשוט Fun.
eval של call - E2 יכול להיות פונקציה, שהרי פונקציה יכולה להיות ארגומנט (נקרא firstClass). בסופו של דבר המשתמש (בשפה שלנו - כך החליט ד"ר ערן) צריך לקבל מספר, על אף שברקט ניתן להחזיר פונקציה.

נדבר כעת על המימוש עצמו - eval:

```
(: eval : FLANG -> FLANG)                          ; <- note return type
  ;; evaluates FLANG expressions by reducing them to *expressions*
  (define (eval expr)
    (cases expr
      [(Num n) expr]                                ; <- change here
      [(Add l r) (arith-op + (eval l) (eval r))] ; <- change here
      [(Sub l r) (arith-op - (eval l) (eval r))] ; <- change here
      [(Mul l r) (arith-op * (eval l) (eval r))] ; <- change here
      [(Div l r) (arith-op / (eval l) (eval r))] ; <- change here
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
```

```
                (eval named-expr)))]          ; <- no `(Num ...)'
      [(Id name) (error 'eval "free identifier: ~s" name)]
      [(Fun bound-id bound-body) expr]          ; <- similar to `Num'
      [(Call (Fun bound-id bound-body) arg-expr) ; <- nested pattern
       (eval (subst bound-body                  ; <- just like `with'
                    bound-id
                    (eval arg-expr)))]
      [(Call something arg-expr)
       (error 'eval "`call' expects a function, got: ~s" something)]))
```

כעת - נחזיר רק flang, בנוסף - עבור כל פעולה מתמטית - ניצור פונקציה שתקבל את הביטויים - שיורידו את ה:flang, תעשה את הפעולה ותעטוף שוב ב:flang. בנוסף: היא תוודא שמדובר רק ב:Num. פונקציה זו הינה arith-op:

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
  ;; gets a Racket numeric binary operator, and uses it within a FLANG
  ;; `Num' wrapper (note H.O type)
  (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))))
```

חשוב לשים לב - היא מקבלת פונקציה של רקט, (+ ... ...) למשל.
נגדיר בפנים פונקציה שפותחת את העטיפה, ומדוע בפונקציה נוספת? בשביל לוודא שהטיפוס הינו נכון (יותר נקי לעשות זאת כך).

נחזור ל:eval: with: נשאר אותו דבר. רק שאין צורך לעטוף ב:Num.
Fun - חוזר כמו שהוא, וכמו שאמרנו לעיל.

אך גם כאן יש באג... (הבאג נגרם כתוצאה מכך שאנחנו מכריחים את Call לקבל Fun בלבד, והדבר יוסבר בהרחבה בהמשך) נריץ ונבין מדוע. נבנה פונקציה להרצה, ונתיר להחזיר רק Num:

```
(: 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)])))
```

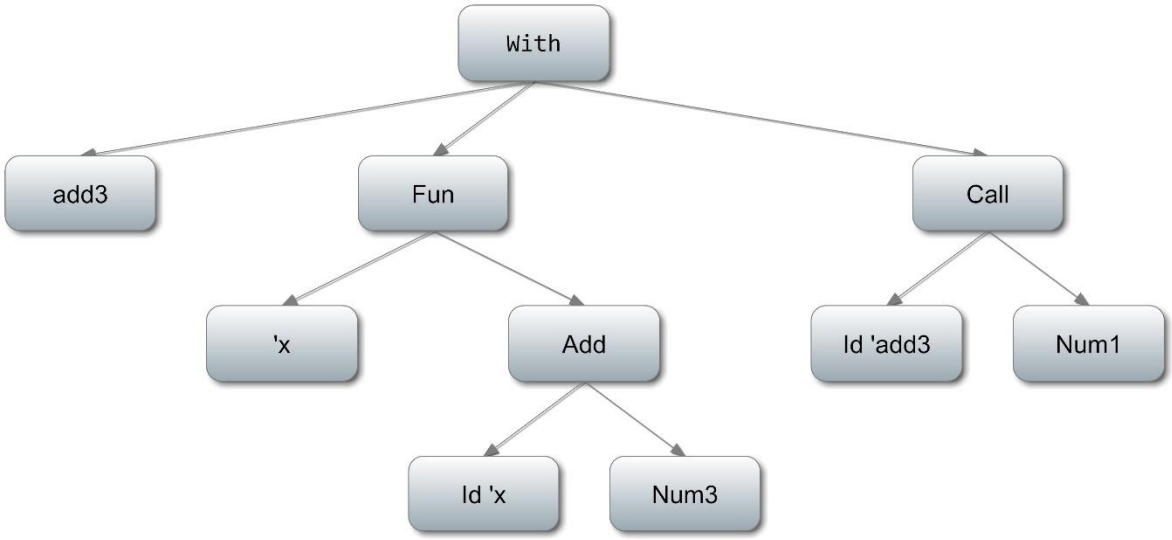
במהלך השיעור העתיק ד"ר ערן את הטסטים והריץ את הראשון:

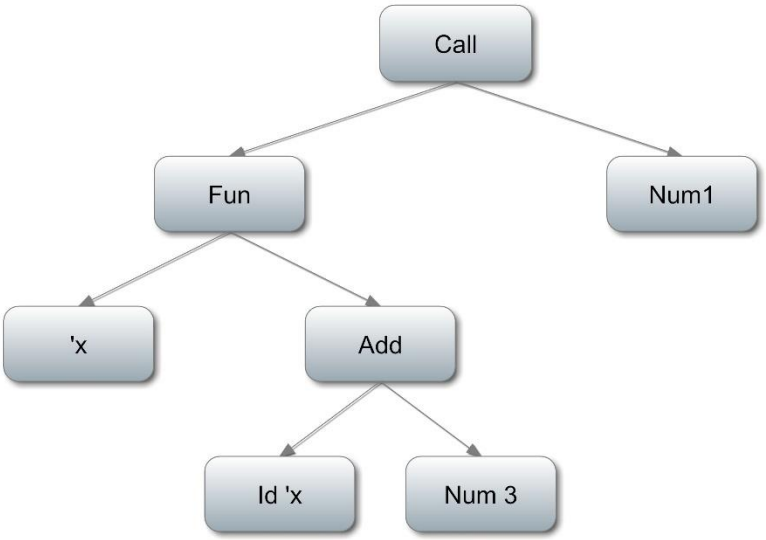
```
(test (run "{call {fun {x} {+ x 1}} 4}")
        => 5)
```

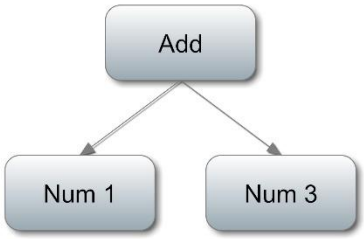
עבור הטסטר השני:

```
(test (run "{with {add3 {fun {x} {+ x 3}}}
                {call add3 1}}")
        => 4)
```

בנינו עץ:



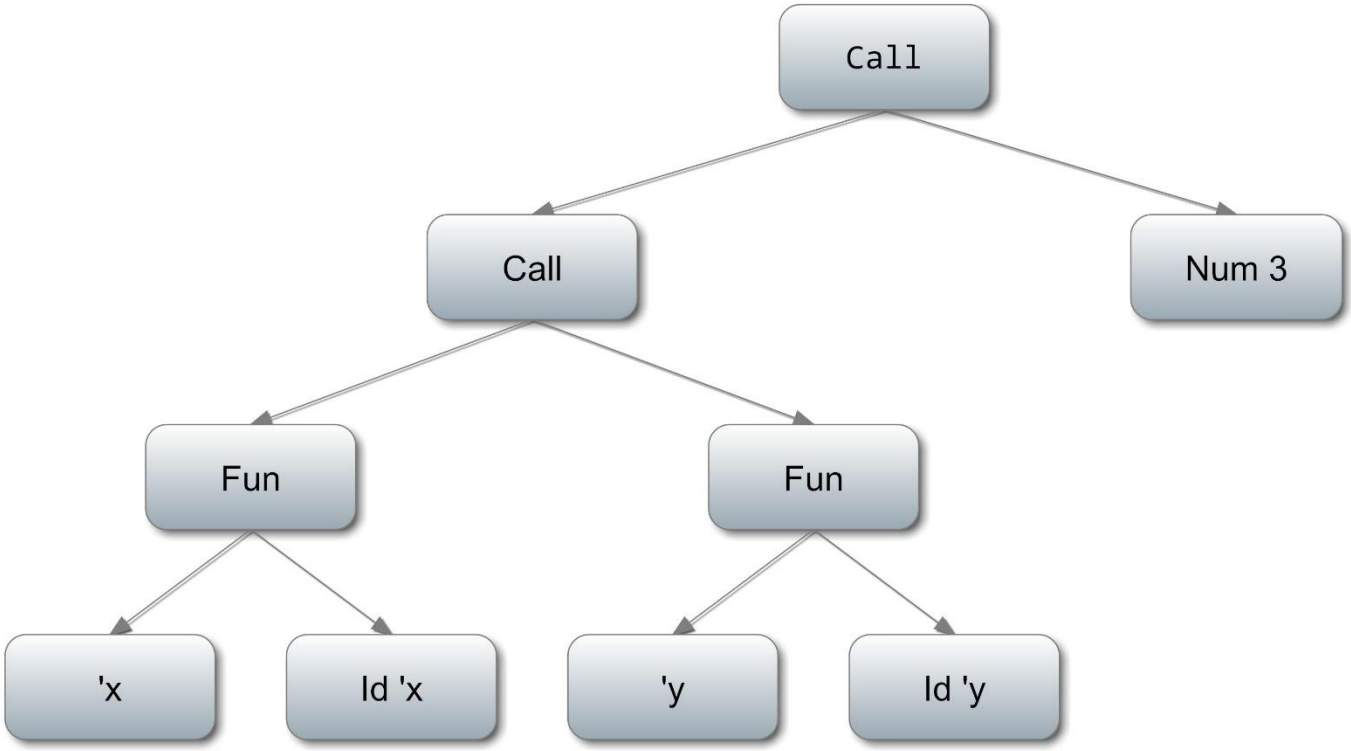




בעצם - נתנו שם לפונקציה. עד כאן הכל תקין. אם כן, היכן הבאג? נשים לב לדוגמא הבאה:

```
{call
    {call
        {fun {x} x}
        {fun {y} y}}
    3}
```

בעצם אני רוצה להפעיל את כל הפונקציה על 3, אך לשם כך - יש לבדוק את הערך של הפונקציה - שזה קורה ע"י הפעלת הפונקציה על הפונקציה, בעצם מ:y חוזרת פונקציה שמחכה ל:y, ומחזירה את y כאשר נותנים לה אותו, ובעצם 3 יתקבל בתור y ויוחזר 3. נעשה לעניין parse , ונריץ את eval.

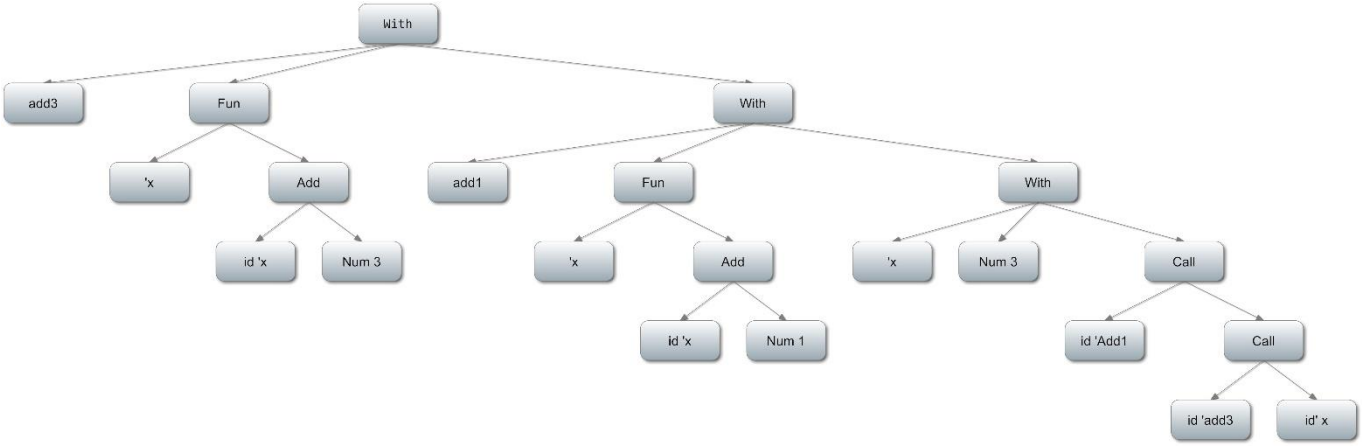


אנחנו בעצם תקועים, כי Call לא חייב לקבל Fun בלבד, אך אם נעשה eval על הביטוי - כן נקבל Fun בלבד.

בשביל שנראה שהבנו את הקוד (ללא קשר לבאג שדיברנו עליו לעיל) ניקח כעת קוד:

```
(test (run "{with {add3 {fun {x} {+ x 3}}}
                {with {add1 {fun {x} {+ x 1}}}
                  {with {x 3}
                    {call add1 {call add3 x}}}}}")
        => 7)
```

ונריץ אותו:



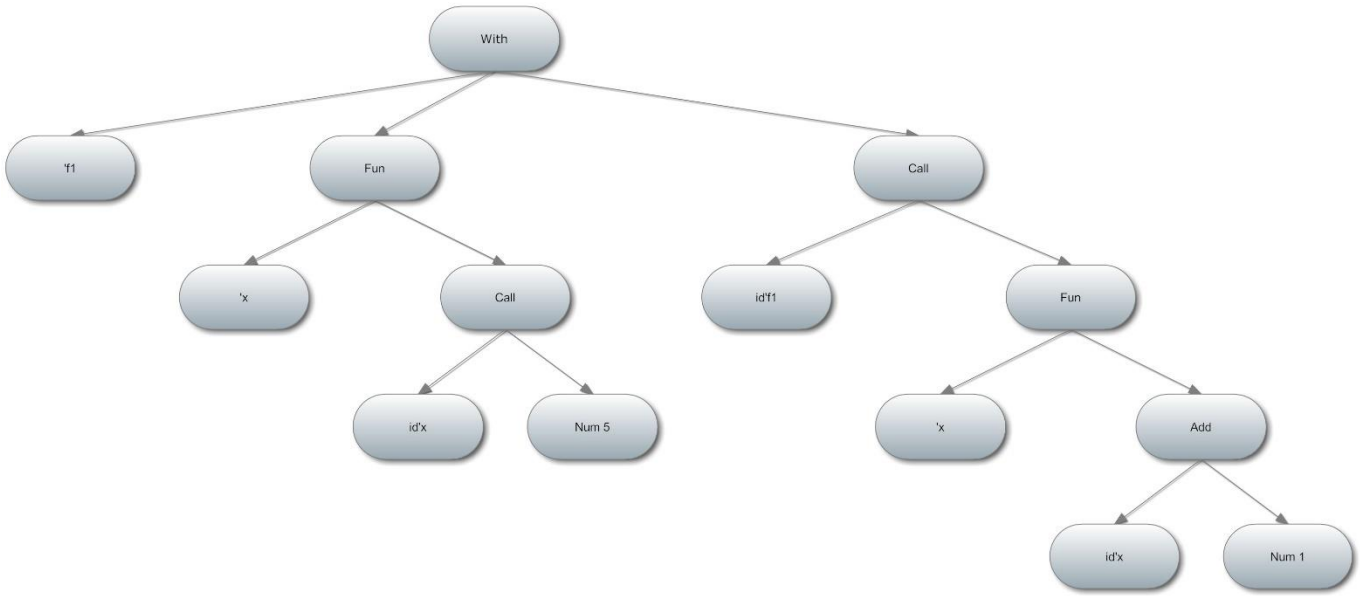
נחזור לבאג: בעצם התיקון יהיה ב:Eval של Call:

```
[(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr)])        ; <- need to evaluate this!
         (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)]))]
```

בעצם אנחנו מגיעים למצב שתמיד נקבל Fun, ז"א: גם אם כרגע קיים Call, בסופו של תהליך נקבל Fun, וזה תקין - אחרת נאמר שיש כאן טעות.

לפני שנמשיך בחומר - נעשה חזרה על הרצה שלימה.
חשוב לשים לב - הfun הראשון מקבל פונקציה והשני מספר

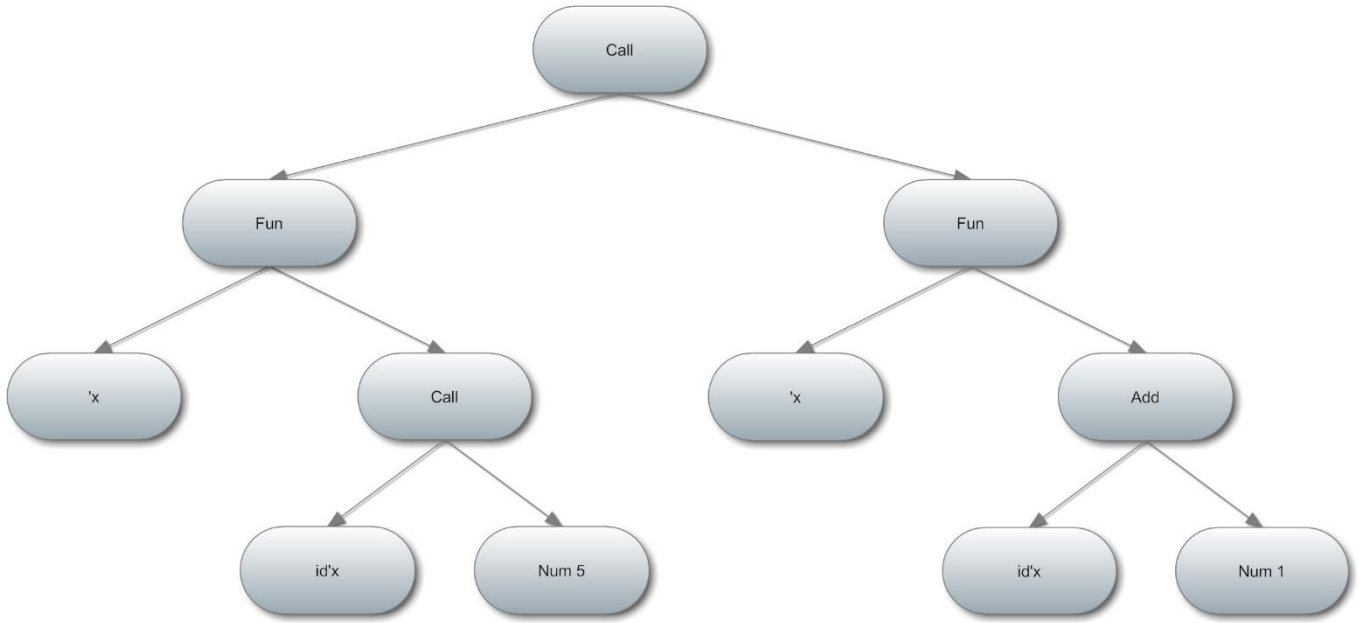
```
{with {f1 {fun {x} {call x 5}}}
     {call f1 {fun {x} {+ x 1}}}}
```



כעת נפעיל Eval:
נלך לפי השלבים:

v = eval(fun) ->fun

בעצם לא עשינו שום שינוי, כעת נעשה subst על ה:body, ולכן שלב ראשון:

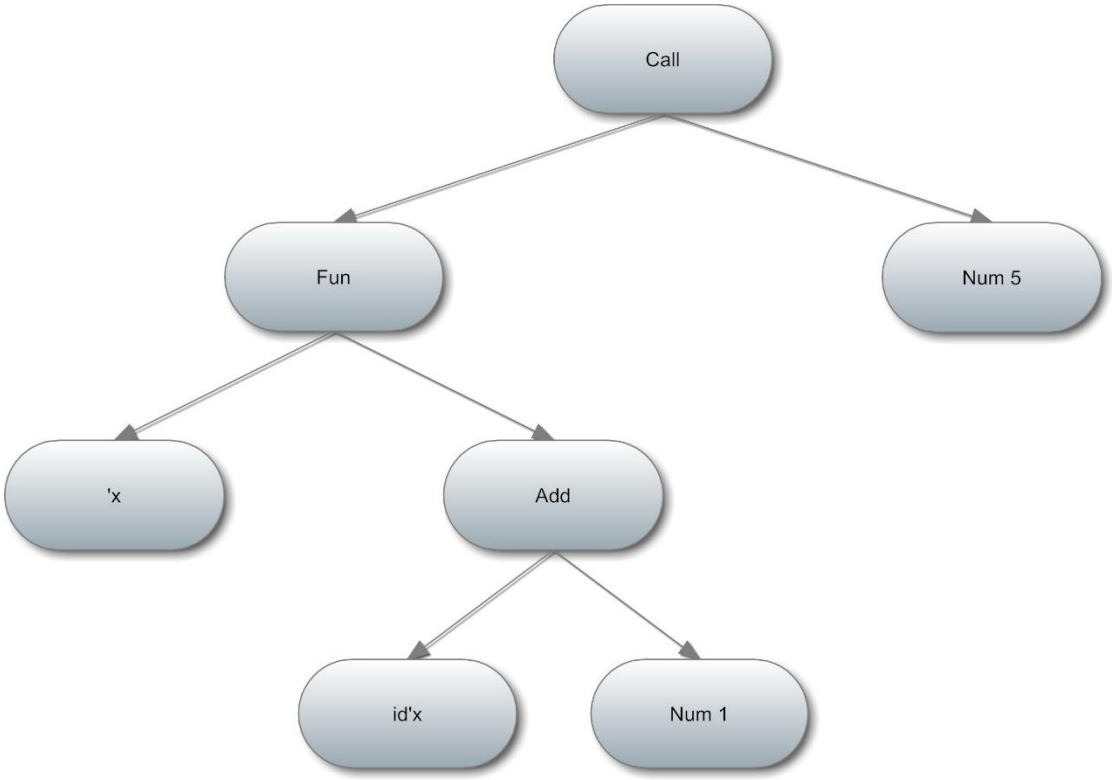


שלב שני - subst על הצד השני - ובעצם לא התרחש שום שינוי.

שלב שלישי - חזרנו מה:subst, נעשה Eval על call (Body של ה:With) - תחילה נעשה בדיקה כי מדובר ב:Fun, במקרה שלנו - בוודאי. כעת - נחזור על השלבים דלעיל:

Eval (arg) = fun -> fun

כעת נעשה subst - בעצם מימין לשמאל: ונקבל:



כעת נעשה

Eval(fun) -> fun
Eval(Num 5) -> Num 5
subst(Add) - > Num 5, Num 1

עד כאן הכל עובד, אך יש כאן חוסר יעילות, שהרי בכל פעם אנחנו עושים ניתוח סינטקטי עבור כל הקוד - וייתכן שהשימוש באותו משתנה - אינו שכיח כמעט. נרצה כעת לעשות את ההחלפה תוך כדי ריצה, במקום ש:id יחזיר Error - נאמר לתוכנית לחפש אותו ב"רשימת ההחלפות".

אם כן - עד עכשיו עבדנו עם subst ורק אחרי זה eval, נעבור כעת למודל אחר: שמירה של ליסט של החלפות.
נגדיר מבנה נתונים חדש:

```
(define-type SubstCache = (Listof (List Symbol FLANG)))
```

בעצם ה:Flang - הוא או פונקציה או מספר (כאשר המספר - יהיה החישוב של Add למשל).

נגדיר גם SubstCache ריק:

```
(: empty-subst : SubstCache)
  (define empty-subst null)
```

בנוסף - נרצה להכניס איברים - כמו מחסנית ולכן נממש פונקציה - extend:

```
(: extend : Symbol FLANG SubstCache -> SubstCache)
  (define (extend id expr sc)
    (cons (list id expr) sc))
```

בנוסף: נרצה לעשות lookup - למצוא איבר ברשימה:
אם הרשימה ריקה - כמובן שלא מצאתי את המבוקש.
אם אינה ריקה - נבדוק אם הראשון מתאים - אם כן - סיימנו - ואני מחזיר את השני של הראשון - כי הרשימה נראית כך:

(…('x (Num 5))…)

כאשר בדוגמא הזו - מחזירים Num 5.

```
(: lookup : Symbol SubstCache -> FLANG)
```

```
(define (lookup name sc)
  (cond [(null? sc) (error 'lookup "no binding for ~s" name)]
        [(eq? name (first (first sc))) (second (first sc))]
        [else (lookup name (rest sc))]))
```

האפשרות למצוא איבר בזוג סדור מתוך רשימה - קיים ברקט: assq, כאשר - היא מחזירה את הזוג הסדור הראשון שהאיבר הראשון זהה, אחרת - false.

נמשש את lookup בעזרת assq:

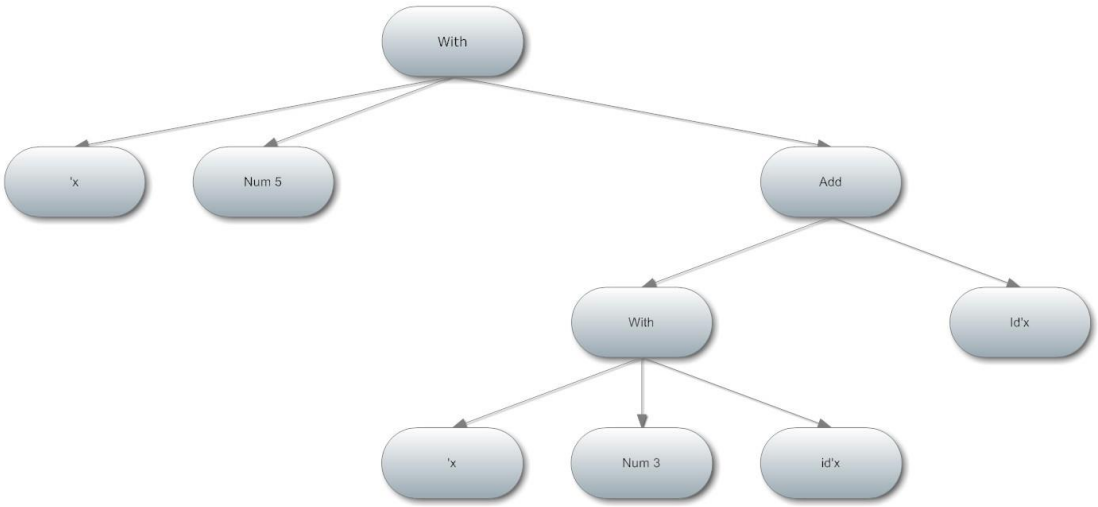
```
(: lookup : Symbol SubstCache -> FLANG)
  (define (lookup name sc)
    (let ([cell (assq name sc)])
      (if cell
        (second cell)
        (error 'lookup "no binding for ~s" name))))
```

נדבר כעת על eval ללא - subst, אלא עם cache. בעצם נשתמש בפונקציות שיצרנו ב:eval דווקא (להכניס ולהוציא). להוציא - רק כאשר הגעתי לבנאי ID בעץ. מתי נכניס? כאשר אנחנו ב:with או call.

```
eval(N,sc)                  = N
eval({+ E1 E2},sc)          = eval(E1,sc) + eval(E2,sc)
eval({- E1 E2},sc)          = eval(E1,sc) - eval(E2,sc)
eval({* E1 E2},sc)          = eval(E1,sc) * eval(E2,sc)
eval({/ E1 E2},sc)          = eval(E1,sc) / eval(E2,sc)
eval(x,sc)                  = lookup(x,sc)
eval({with {x E1} E2},sc)   = eval(E2,extend(x,eval(E1,sc),sc))
eval({fun {x} E},sc)        = {fun {x} E}
eval({call E1 E2},sc)
          = eval(Ef,extend(x,eval(E2,sc),sc))
                             if eval(E1,sc) = {fun {x} Ef}
          = error!           otherwise
```

חשוב לשים לב - לגבי ה:eval של call - בעצם אנחנו שואלים אם ה:if מתקיים, אם הוא אכן מתקיים - אנחנו מפעילים את ה:eval הזה: `eval(Ef,extend(x,eval(E2,sc),sc))`

לדוגמא העץ:



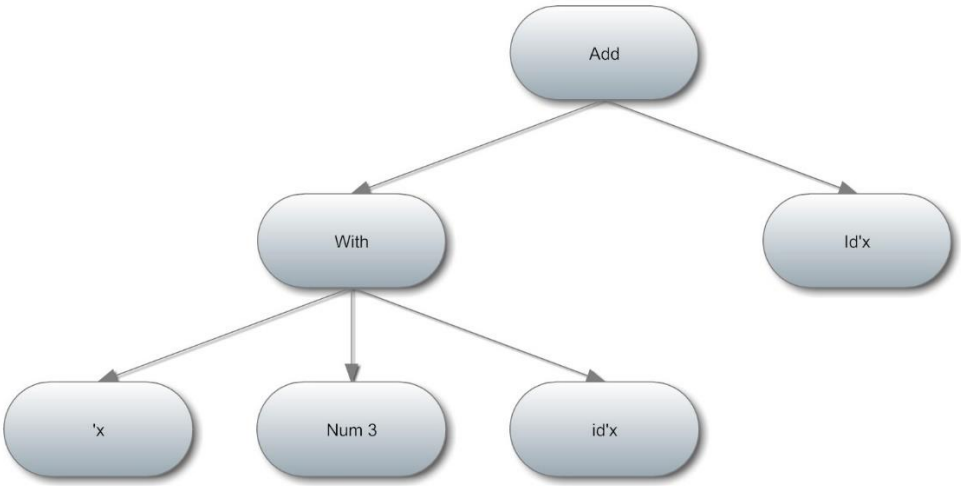
נעריך תחילה את האמצע - בעצם נקבל Num 5. כרגע הרשימה sc ריקה, אך לאחר ה:with נעשה:

(extend (list 'x (Num 5)), sc)

בעצם נקבל:

sc = '(x (Num 5))

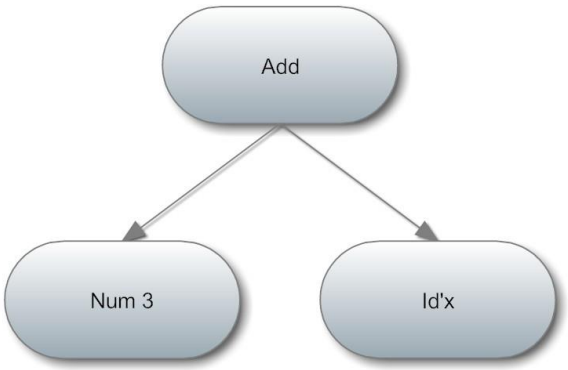
כעת אפשר להוריד את תחילת העץ ולהמשיך עם Add, ולכן:



כעת יש לעשות eval על 2 הבנים של Add, כמו שהינו רגילים עד עכשיו, רק שכעת - נשלח גם את sc בצורה רקורסיבית. נתחיל עם הבן השמאלי: הגענו ל:with, ולכן נפעל באותו אופן שפעלנו למעלה: נעשה eval על num 3 ונקבל num 3. וכעת נכניס את הזוג הסדור ('x (num 3)) לתוך ה:sc:

(extends ('x (Num 3) sc) -> '((x (Num 3)) (x (Num 5)))

כעת נעבור ל:body של with - שם אנחנו נתקלים בבנאי של ID ולכן - נשלוף מתוך המחסנית sc את האיבר המתאים (לפי חוקי מחסנית - ז"א: האיבר הראשון). ולכן נחזיר Num 3 כך שהעץ יראה באופן הבא:



כעת נשאר לעשות eval על הצד הימני של העץ, ושוב - הגענו לבנאי מסוג ID - כל מה שנותר לעשות - להחזיר מתוך SC את האיבר המתאים.

ובעצם קיבלנו ((Add (Num 3) (Num 5)).

נראה כעת את הקוד עצמו:

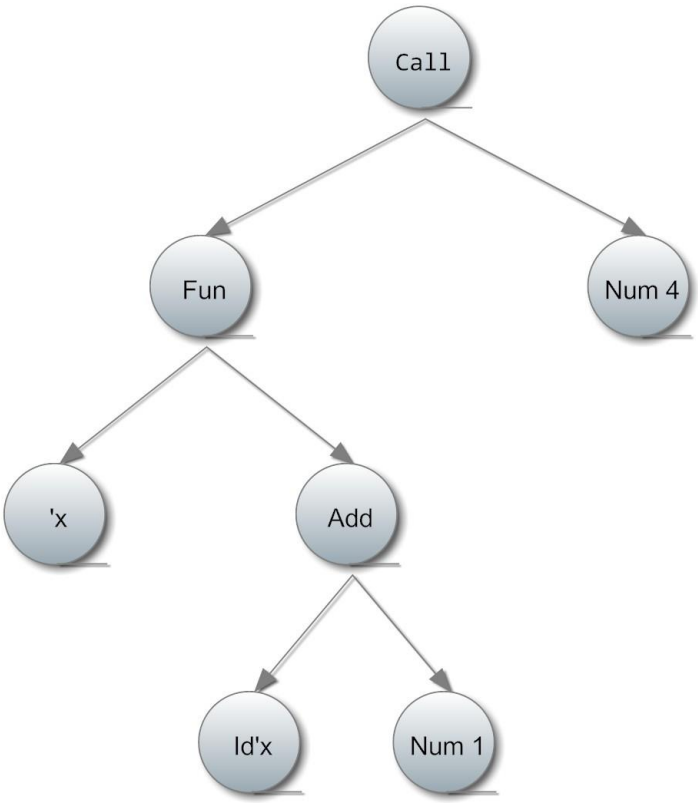
```
(: eval : FLANG SubstCache -> FLANG)
  ;; evaluates FLANG expressions by reducing them to expressions
  (define (eval expr sc)
    (cases expr
      [(Num n) expr]
      [(Add l r) (arith-op + (eval l sc) (eval r sc))]
      [(Sub l r) (arith-op - (eval l sc) (eval r sc))]
      [(Mul l r) (arith-op * (eval l sc) (eval r sc))]
      [(Div l r) (arith-op / (eval l sc) (eval r sc))]
      [(With bound-id named-expr bound-body)
       (eval bound-body
             (extend bound-id (eval named-expr sc) sc))]
      [(Id name) (lookup name sc)]
      [(Fun bound-id bound-body) expr]
      [(Call fun-expr arg-expr)
       (let ([fval (eval fun-expr sc)])
         (cases fval
           [(Fun bound-id bound-body)
            (eval bound-body
                  (extend bound-id (eval arg-expr sc) sc))]
           [else (error 'eval
```

```
  "`call' expects a function, got: ~s"
fval)]))]))
```

תחילת השיעור עשה ד"ר ערן חזרה על החומר... והריץ דוגמאות:

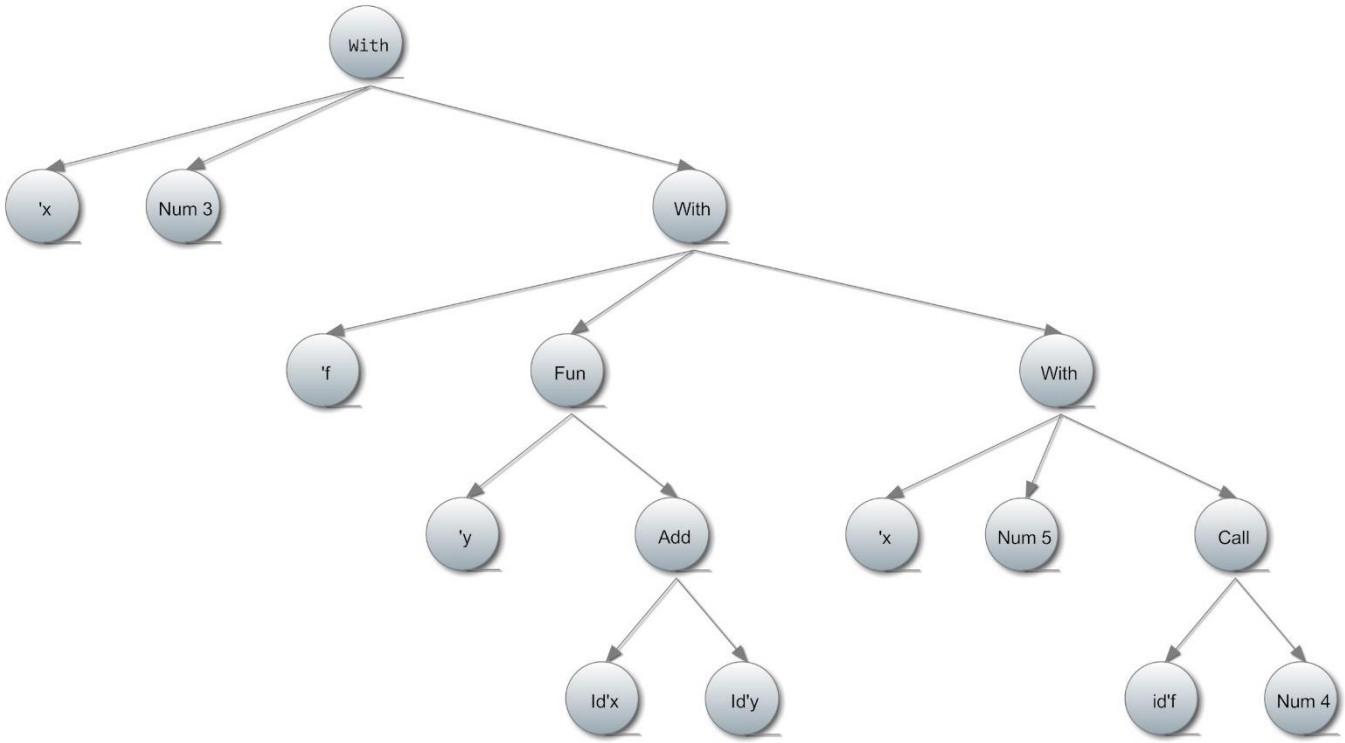
{call {fun {x} {+ x 1}} 4}

```
1. eval(call)  sc0 = ()
2. eval(Fun sc0) -> fval
3. eval(Num 4 sc0) -> Num 4
4. eval(Add (extend x (Num 4) sc0)
5. eval(Id'x sc1)-> lookup 'x sc1 -> Num 4
6. eval(Num 1 sc1) -> Num 1
7. + 1 4 -> 5
```



נעבור לדוגמא המסומנת בצהוב: לכאורה - היה צריך לחזור 7, על פי מודל ה:subst:

```
{with {x 3}
      {with {f {fun {y} {+ x y}}}
            {with {x 5}
                  {call f 4}}}}
```



כעת, נבדוק לפי המודל החדש:

```
sc0 = '()
eval({with {x Num 3} BoundBody},sc0) =
                    eval(BoundBody, extend(x,eval(Num 3,sc0), sc0))]
                    -> sc1 = '((x Num 3))
eval({with {f BoundBody } Fun},sc1) =
                    eval(BoundBody, extend(f,eval(Fun,sc1), sc1))]
                    -> sc2 = '((f Fun) (x Num 3))
eval({with {x Num 5} BoundBody},sc2) =
                    eval(BoundBody, extend(x,eval(Num 5,sc2), sc2))]
                    -> sc3 = ((x Num 5) (f Fun) (x Num 3))
eval({call id'f Num 4},sc3)
```

שלב מקדים: נעשה eval של:

```
eval(id'f,sc3) = lookup(id'f)   -> {fun {y} {+ x y}}
                                -> Ef = {+ x y}
                  = eval(Ef,extend(y,eval(Num 4,sc3),sc3))
                  -> sc4 = ((y Num 4) (x Num 5) (f Fun) (x Num 3))
```

כל מה שנותר לעשות זה: eval({+ x y},sc3), ז"א:

```
eval({+ id'x id'y},sc) = eval(id'x,sc4) + eval(id'y,sc4)
eval(id'x,sc4) -> lookup(x) = Num 5
eval(id'y,sc4) -> lookup(y) = Num 4
```

אכן קיבלנו 9, במקום 7. הבעיה היא ב:eval על fun. בעצם - הוא צריך לזכור את כל ההחלפות שנעשו עד עכשיו - והוא לא צריך להכיר את הסימבול שקיים בהמשך. בעצם בפונקציה אסור שיהיו מופעים חופשיים.

במהלך השיעור הראה ד"ר ערן שיש מעין overloading להגדרה של משתנים (ראה דוגמא של "מס"), וכל זה רק עבור שיטה בה רקט עובדת, מה שאין כן בשפות שאנחנו מכירים. מה שמקשר אותנו - לשמור את כל ה:subs-catch שהיו עד עכשיו, כי אנחנו לא רוצים שהמשתנים ידרסו.

ד"ר ערן דילג על ההקדמות, עד Implementing Lexical Scope. בעצם נגדיר סוג של ענן נוסף - ענן של סביבה, בו נשמור את מה שהיה עד עכשיו, ולכן עבור Fun נוסיף sc, ועבור Call נוסיף עוד sc.

בסוף השיעור התחיל ד"ר ערן לדבר על השינויים בגוף הקוד, אך הדברים לא היו ברורים, נבקש שיחזור על הנקודה הזו בתחילת השיעור.

נחזור על הנקודה האחרונה משיעור שעבר - אנחנו נרצה גם לזכור את הסביבה בזמן שמערכים את הפונקציה. נגדיר אובייקט חדש שיודע לשמור זאת.

לכן פונקציה כבר לא יכולה להיות FLANG, אלא משהו גדול יותר, והשוני יבוא לידי ביטוי בכך שכאשר נעשה הערכה לפונקציה - תשלח גם הסביבה. הכרת הסביבה תתרחש רק בזמן הריצה - נגדיר משתנה חדש בשם ENV: (אנחנו כבר לא צריכים יותר SC).

```
(define-type ENV
    [EmptyEnv]
    [Extend Symbol VAL ENV])
```

בנאים: סביבה ריקה, וסביבה שמחברת סביבות.

נעשה גם lookup בעזרת cases:

```
(: lookup : Symbol ENV -> VAL)
  (define (lookup name env)
    (cases env
      [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
      [(Extend id val rest-env)
       (if (eq? id name) val (lookup name rest-env))]))
```

אך מיהו val? משתנה חדש: כי כל ה:eval משתנה, וזה הטיפוס החדש של eval:

```
(define-type VAL
    [NumV Number]
    [FunV Symbol FLANG ENV]) ; arg-name, body, scope
```

כך נוכל ליצור מעטפת למספרים ולפונקציות.

נעבור ל:eval:

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

ה:eval מקבל FLANG וסביבה ENV, ומחזיר val.

אם מספר מגיע - אני יעטוף אותו ב:NumV, בדיוק כמו ב:NUM
הפונ' הארית' נשארות אותו דבר, יש רק לשנות את arith-op:

```
(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
  ;; gets a Racket numeric binary operator, and uses it within a NumV
  ;; wrapper
  (define (arith-op op val1 val2)
    (: NumV->number : VAL -> Number)
    (define (NumV->number v)
      (cases v
        [(NumV n) n]
        [else (error 'arith-op "expects a number, got: ~s" v)]))
    (NumV (op (NumV->number val1) (NumV->number val2))))
```

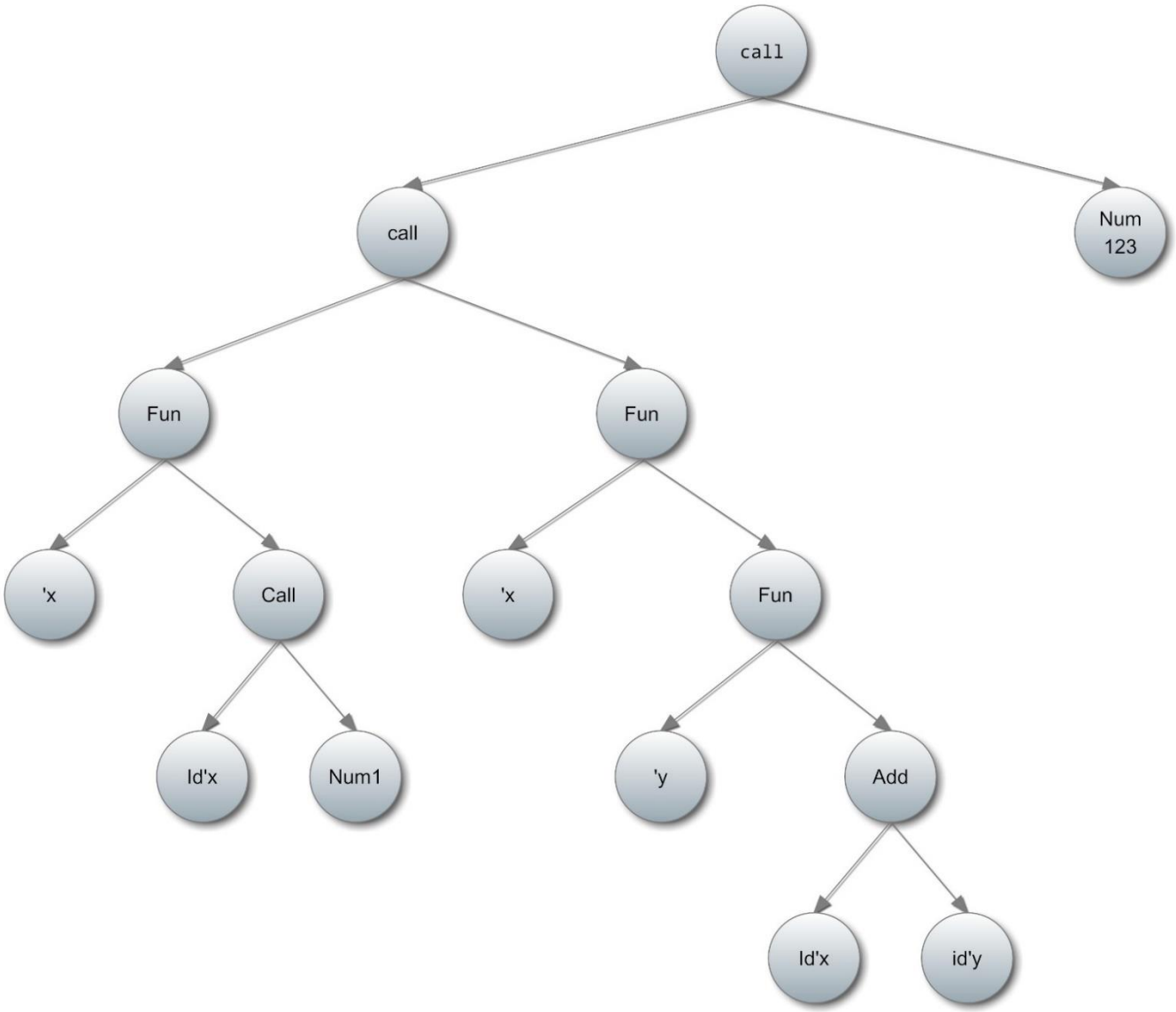
כיצד נעשה eval ל:with? - בדיוק אותו דבר כמו שהיה מקודם.

גם fun פשוט מאוד - נשתמש בסביבה הנוכחית, וזה מה שתכיר הפונקציה.

בסוף השיעור הריץ ד"ר ערן דוגמא מהקוד:

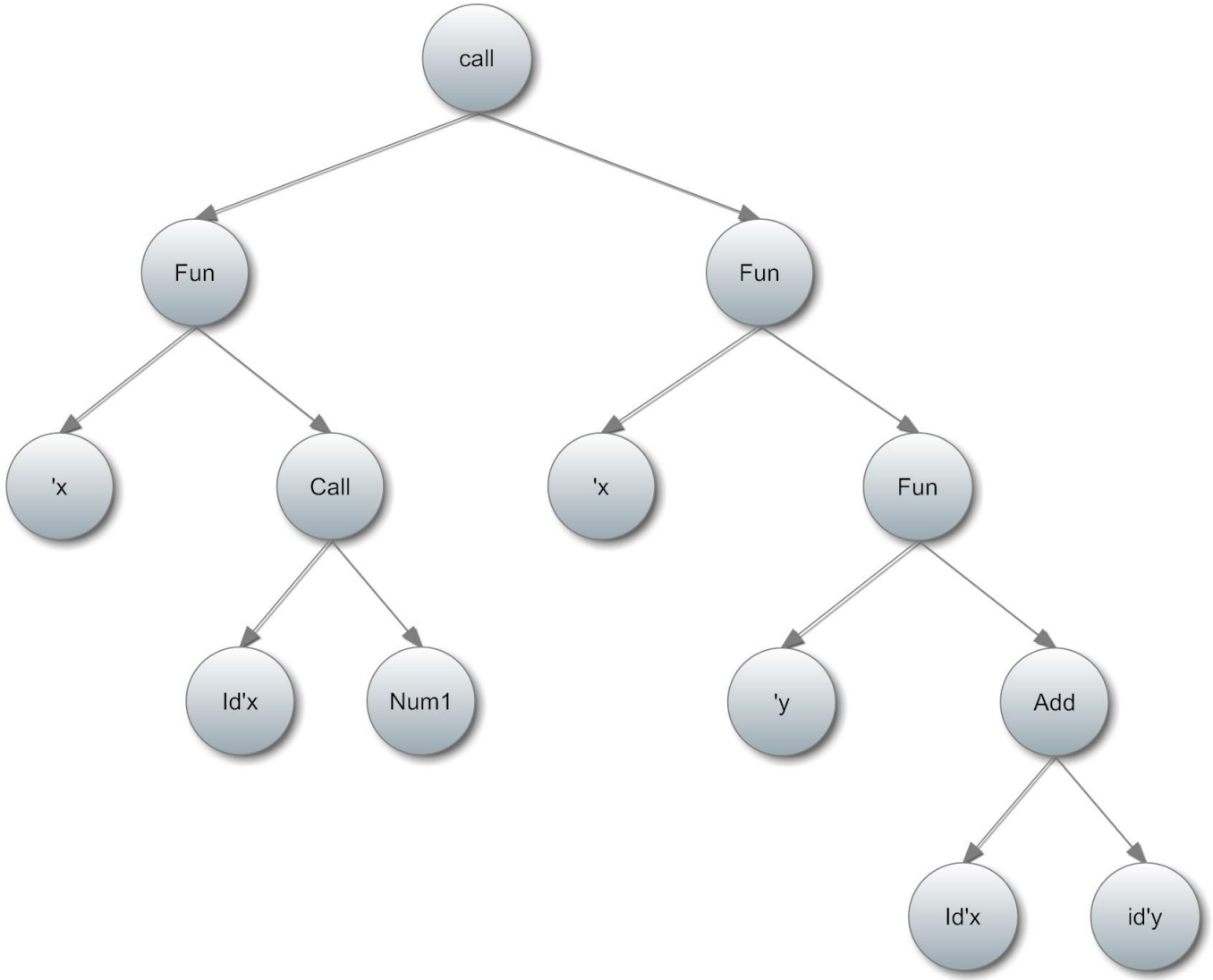
```
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
```

נצייר את הדוגמא:

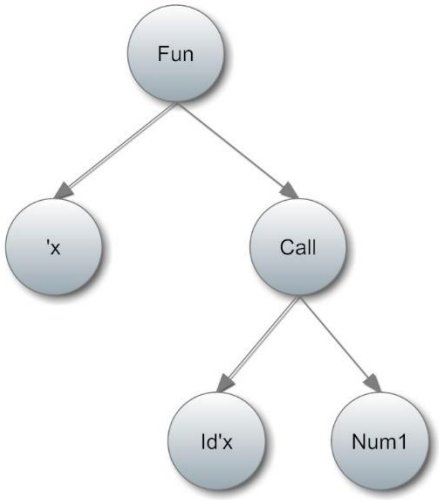


דוגמא זו לא הייתה עובדת ב:SC, נראה זאת: (להשלים את הדוגמא)...

Ast1 =
Res1 =
SC1 = E



Ast2 =
Res2 =
SC2 = E



Ast3 =
Res3 = (Fun x (Call id'x (Num 1)))
SC3 = E

לסיום: קיים באג נוסף שלא שמנו לב אליו - ב:subst, אך גם לא נתקן אותו:

```
(run "{with {f {fun {y} {+ x y}}}
          {with {x 7}
            {call f 1}}}")
```

כמובן שאמור להיות כאן טעות - כי x אינו ידוע, אך מה הינו מקבלים בהפעלת subst רגיל? כאשר נחפש את id'x - הוא כבר הוחלף...!