שפות תכנות

3 ביולי 2020

על פי הרצאותי של ד"ר עמרי ערן

להארות/תיקונים - נעם דומוביץ , פאל: 0508752542

תוכן עניינים

- מרצה: ד"ר עמרי ערן
- בחינה : 80% מטלות 20% , צריך לעבור בשניהם.
- מטלות חלקן ביחידים וחלקן בזוגות

לינקים שימושיים:

- האתר של ברזילי: `https://pl.barzilay.org/resources.html`
- אוסף פונקציות: `https://docs.racket-lang.org/racket-cheat/index.html`

0.1 הקדמה מהן יסודות של השפה?

- תחביר $(Syntax)$ - מערכת הכללים של השפה.
- סמנטיקה $(Semantics)$ משמעויות. כגון: משמעות העומדת מאחורי מילה מסוימת בשפה.

0.1.1 הבדל בין תחביר לסמנטיקה

- תחביר חשוב בעיקר לצורכי "קוסמטיקה". הוא נועד לשרת את הסמנטיקה. <u>סמנטיקה חשובה יותר מהתחביר</u>. שכן, תחביר אפשר לשנות.
- לדוג':
  - אם נסתכל על מס' 24 שכתוב בקובץ כלשהו- מדובר בעצם בייצוג של שני ערכי IICSA ,לעומת המס' 24 שמאוחסן בזיכרון.
  - המילה "שוד" - היא סתם מילה. אבל אם נחשוב על המשמעות שלה - אפשר לשבת בכלא בגלל ביצוע שוד.
  - דוגמה נוספת המילה 'חמור' - נוכל להחליט שמהיום היא 'למור' והיא מתייחסת לאותה חיה ולאחר זמן "הסתגלות", כולנו נתרגל למילה החדשה
  - דוגמה מעולם שפות התכנות - ניקח אוסף הצהרות זהות משפות שונות:
  - a[25]+5 (Java:)
  - (+ (vector-ref a 25) 5) (Racket:)
  - a[25]+5 (JavaScript:)
  - a[25]+5 (Python:)
  - $a[25]+5 (Perl:)
  - a[25]+5 (C:)
  - a[25]+5 (ML:)
    - מבחינה סינסטקטית באמת ניתן לראות שכל שפה יש לה את המיוחד שלה (וראקט יוצאת דופן)
    - מבחינה סמנטית כולן מקפיצות $exception$ חוץ מ C שבה לא מוגדר מה יקרה

    ניתן להוסיף שזה גם מאוד מתאים לאג'נדה של כותבי השפה, באותם ימים היו מעט מאוד מתכנתים ורובם היו ברמה גבוה מאוד , והם רצו לתת מה שיותר כח למתכנתים. כיום תכנות הוא יותר רווח ואנחנו רוצים להגן על המתכנתים.
- איך נגדיר סמנטיקה של שפה?

- אפשרות אחת בעזרת חוקים לוגים
- על ידי תרגום לשפה אחרת, כלומר על ידי כלי שמתרגם אנחנו יוצקים משמעות לתוכן הכתוב, וזהו בעצם קומפיילר (אנחנו נכתוב אינטרפטר)
  * לכן נבחרה $Racket$ שהיא מתאימה לעקרונות של הקורס הזה.
  * נעיר שלמרות ש $Racket$יוצאת דופן יש להתייחס לזה ברצינות (הסיפור עם דיקטסרה שטען שצריך לותר על $Goto$

0.2 $Racket$

0.2.1 בסיס

- זוהי שפה פונקציונלית
  - שפה שמתייחסת לעולם כאבסטרקציה, ללא קשר לאיך הם ממומשים בזיכרון, והקוד "מסביר את עצמו" כי נדאג שלא יהיה $side-effect$ ותמיד יהיה ערך מוחזר לעבוד איתו (ולא יהיה לנו דברים שקיימים מבחינתנו רק בזיכרון)
- $Racket$ שפת בת ל $Scheme$ (התפצלו לאחר ריב בחברה)
- כל קובץ נתחיל צריך להתחיל בסוג השפה שאני משתמש אצלנו ב:

#lang pl

- הטיפוסים דינמיים - f יכולה להיות פונקציה ויכולה להיות מחרוזת.
  - בפיתוחים החדשים של השפה כבר כן חייבים להגדיר
  - עולם הערכים , הוא אבסטרקטי והיררכי למשל:
    * ישנו אובייקט/ערך $true$ וישנו אובייקט/ערך $false$ ואיחוד הערכים יוצר את האובייקט/ערך $boolean$
- בעולם של $Racket$ האובייקט היחידי שהוא $false$ ולכן כל דבר אחר למשל כל מספר (ואפילו 0) הוא $true$
- באופן דומה עבור מספרים .
- טיפוס $string$ נסמן על ידי ".."
- טיפוס $Symobl$ נסמן על ידי תו בודד ' (רק בהתחלה)
- תווים $Characters$: אובייקט למשל התו a יסומן כך: $\#\backslash a$
- וכעת ניתן לעשות פעולות, על ידי הצבת אופרטור/פונקציה משמאל כמו שאנחנו מסמנים $f(x,y)$, לדוג':

(string #\a #\b)

- $null$ הוא מסוג רשימה ריקה
- מוסכמה: פונקציות בוליאניות יסומנו ב? בסופן למשל:

(eq? #\a #\a) - eq ask if a==a , retrun bool/

הרצאה 2

Commetns

מסמנים על ידי ; המוסכמה לעשות על ידי פעמיים כלומר ;; דוגמה:

```
;; This is a comment that continues to
;; the end of the line.
; One semi-colon is enough.
```

להגדיר בלוק כהערה על ידי |# בשביל לפתוח ו#| בשביל לסגור (להתחיל מחדש)

```
#| This is a block comment , which starts
    with '#|' and ends with a '|# '.
|#
```

ביטוי בודד על ידי #;

```
#;( comment out a single form )
```

0.2.2 ביטויי תנאי

```
There are two Boolean values built-in in Racket:
'#t' (true) and '#f' (false).
```

if :

$$(\underbrace{if}_{\text{oper'}}\underbrace{(<\ 2\ 3)}_{\text{cond'}}\underbrace{10}_{\text{true}}\underbrace{20}_{\text{false}}) - - > 10$$

דוגמה:

$$(\underbrace{if}_{\text{oper'}}\underbrace{(\text{"}false\text{"}}_{\text{cond'}}\underbrace{1}_{\text{true}}\underbrace{2}_{\text{false}}) - - > 1$$

$$(\underbrace{if}_{\text{oper'}}\underbrace{null}_{\text{cond'}}\underbrace{1}_{\text{true}}\underbrace{2}_{\text{false}}) - - > 1$$

הסבר: ה$string$, $\text{"}false\text{"}$ ו $null$ הם $\#t$ כפי שהסברנו לעיל. רק:

$$(\underbrace{if}_{\text{oper'}}\underbrace{\#f}_{\text{cond'}}\underbrace{1}_{\text{true}}\underbrace{2}_{\text{false}}) - - > 2$$

בPL חייבים תמיד להחזיר ערך כלשהו (שפה פונקציאלית ולכן הביטוי הבא אינו תקין:

```
(if test consequent)
```

Cond

התניה מקוננת , ב$Java$ אנחנו מכירים זאת כ $else\ if$, ב$Racket$ זה לא נח ולא מסתדר עם ההזחות, ולכן המציאו את המילה $cond$

תהליך זה נקרא syntactic sugar: תהליך שבו יש לי בשפה פתרון לפעולה מסוימת אבל אני רוצה ל"המתיק" את חיי ולכן ממציא תחביר חלופי שעושה את אותו דבר.

דוגמה:

```
(define (digit-num n)
    (cond [(<= n 9)    1]
          [(<= n 99)   2]
          [(<= n 999)  3]
          [(<= n 9999) 4]
          [else        "a lot"]))
```

סדר הפעולות: בודקים את התנאי הראשון, אם $True$ מחזירים את הערך שלו, אחרת בודקים את התנאי השני, וכו'... בגלל שחייבים להחזיר ערך, **חייב** להיות $else$

0.2.3 רשימות

אובייקט שמגודר באופן רקוריסיבי:

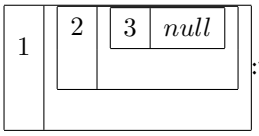
- רשימה ריקה - $null \equiv ()$
- או זוג $(pair)$ שהאיבר השני בו הוא רשימה.

מה זה $pair$? זה זוג שנוצר בעקבות הפקודה $cons$ למשל:

פקודה זו לא ניתן להריץ ב PL , $(\text{cons } 1\ 2) \longrightarrow \boxed{1 \mid 2}$

בPL נגדיר רשימה כך:$(\text{list } 1\ 2\ 3)$

וע"פ ההגדרה המימוש של זה הוא כך:



ניתן להסתכל על זה בצורה יותר מוכרת: $\boxed{1 \mid \cdot} \rightarrow \boxed{2 \mid \cdot} \rightarrow \boxed{3 \mid null}$

לכן הוספת איבר ראשון משמאל , קל מאוד: $(\text{cons } 0\ (\text{list } 1\ 2\ 3))$ (גם ההסרה פשוטה)

הערה: הפקודה (list 1 2 3 null) היא אובייקט שהזוג האחרון בו הוא $\boxed{null \mid null} = \boxed{'() \mid '()}$

<u>פונקציות על רשימות</u>

append

```
( append ( list 1 2) null ) ; = > '(1 2)
( append ( list 1 2) ( list 3 4) ) ; = > '(1 2 3 4)
```

גישה ניתן לגשת על ידי $first, second, third$ לשלושת המקומות הראשונים.

$rest$ זה ההמשך של הרשימה.

$list - ref$ - גישה לפי אינדקס

```
( first ( list 1 2 3) ) ; = > 1
( rest ( list 1 2 3) ) ; = > '(2 3)
( list-ref '(1 2 3) 2) ; = > 3
```

0.2.4 $define$

מאפשר לנו לעשות binding בן שם מזהה לערך, נזכיר שאנחנו מתעסקים ב $Static - typing$ ולכן תמיד צריך להגדיר מהו סוג הערך.

נעשה זאת על ידי נקודתיים לפני ואחרי

דוגמאות:

```
(: PI : Real)
( define PI 3.14)
```

ומעתה יש לנו "קבוע" PI , דוגמה נוספת :

הסבר: השם $length$ מקושר לאובייקט מסוג פונקציה (בגלל החץ) שמקבלת רשימה של Any ומחזירה מספר טבעי

הפוקציה מחזירה את אורך הרשימה

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

למה $Static - typing$?

- לשמור על המתכנת
- עוזר לפונקציה "לכתוב את עצמה"
- הופך את הקוד לקריא יותר.

$define - type$

הזכרנו שניתן להגדיר אובייקט , דוגמה:

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

ניתן להתייחס ל $Snake, Tiger$ כבנאים של האובייקט $Animal$ שמקבלים ערכים ומחזרים אובייקט כזה מסוג $Animl$

כעת גם יש לנו פונקציה $Animal?$ ששואלת האם אובייקט הוא מסוג $Animal$

```
( Animal ? ( Snake 'Slimey 10 'rats ) ) ; = > #t
( Animal ? ( Tiger 'Tony 12) ) ; = > #t
( Animal ? 10) ; = > #f
```

<u>$Cases$</u>

פונקציה שמאפשרת לאובייקט מסוג $Animal$ ולבדוק איך בנינו אותו, לדוגמה:

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

- מי זה n ? הפונקציה מיצרת $binding$ בין הערך לסוג האובייקט ולכן במקרה הזה $n =' Slimey$ $(w = 10, f =' rats)$
- האם צריך $else$ (כמו ב $cond$) ? בגלל שפה אנחנו יודעים בדיוק מי הבנאים שלו אז ניתן לותר עליו, אבל יש מקרים שכן נוסיף

המשך הדוגמה:

```
(: animal-name : Animal - > Symbol )
( define ( animal-name a )
( cases a
    [( Snake n w f ) n ]
    [( Tiger n sc ) n ]) )
```

הסבר:

- קראנו לפונקציה בשם $animal - name$, שלוקחת $Animal$ ומחזירה $Symbol$
- כעת בשביל לדעת מהו האובייקט שקיבלנו נשתמש ב: $Cases$

0.2.5 All

```
(: every ? : ( All ( A ) ( A - > Boolean ) ( Listof A ) - > Boolean ) )
```

הגדרנו מעין $Template$, כלומר אתה לא חייב להחליט כרגע מהוא A אבל כשהחלטת, הפונקציה $every$ תצפה לקבל את אותו אובייקט לשאר הפונקציות

לדוגמה A הוא $Natural$ אז $every$ מצפה לקבל פונקציות מסוג: $\left\{\begin{matrix} Natural \to Boolean \\ ListOf\ Natrual \to Boolean \end{matrix}\right\}$

0.3 דקדוקים חסרי־הקשר

0.3.1 השפה הבסיסית AE

כעת נתחיל לבנות את השפה שלנו, וצריך להתחיל להגדיר את הדקדוק שלה, נניח שאנחנו רוצים להגדיר שפה של מספרים ופעולות חיבור/חיסור. אז התחביר שלנו יהיה לפי 3 כללים:

```
1) <AE> ::= <num>
2)        | <AE> + <AE>
3)        | <AE> - <AE>
```

כלל 1 אומר שמ AE ניתן לגזור כל מספר - זה כאילו הגדרנו אינסוף כללים לגזירת כל המספרים.

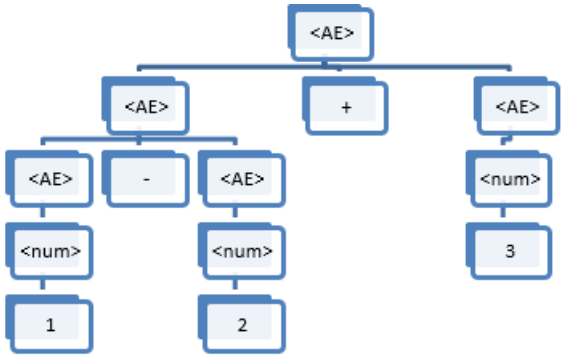
פורמלית ניתן להגדיר, כלל נוסף:

```
<NUM> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
        | <NUM> <NUM>
```

אז מהי השפה? השפה היא כל דבר שניתן לגזור מ $non-terminal$ המרכזי. דוגמה נגזור מAE את $1-2+3$:

```
<AE>          ; ==>
<AE> + <AE>          ; (2) ==>
<AE> + <num>         ; (1) ==>
<AE> - <AE> + <num>  ; (3) ==>
<AE> - <AE> + 3      ; (num) ==>
<num> - <AE> + 3     ; (1) ==>
<num> - <num> + 3    ; (1) ==>
1 - <num> + 3        ; (num) ==>
1 - 2 + 3            ; (num)
```

הבעיה שעץ הגזירה הוא לא בהכרח יחיד, וכאשר אנחנו מילה/ביטוי אנחנו לא יודעים מאיזה עץ היא הגיע. $(Ambiguity)$ -אנחנו לא אוהבים בעולם התכנות.



והבעיה נוספת היא בעיה **סמנטית** - שלמשל בעבור גזירות שונות נקבל תוצאות שונות (סדר הפעולות חשבון) = דו משמעות :

$$-4 = 1 - 2 + 3 = 2$$

לכן נחייב את הדקדוק לגזור את האיבר הראשון למספר, כלומר:

```
1) <AE> ::= <num>
2)        | <num> + <AE>
3)        | <num> - <AE>
```

ובכך יצרנו עץ Left association והוא יחיד ובטלנו את הדו־משמעות.

נמשיך לפתח ־ קדימות לכפל תמומש בכך שכפל יהיה רק בתחתית העץ, ובשביל זה נגדיר:

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

נמשיך לפתח ־ הוספת סוגריים (ואז ביטלנו את הדיון הקודם) שבא מכריחים להגדיר קדימויות, בהתאמה ל*Racket* זה יראה כך:

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

ולמשל :

```
{+ 5 6}--> 11
```

הרצאה 3

בשיעור שעבר הגדרנו את הדקדוק:

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

בעזרת הדקדוק אנחנו בודקים אילו "תוכניות" קבילות בשפה שלנו, נראה בדוגמה:

$$(-\,(+\ 3\ 4)\,2)$$

$$\begin{array}{ccccccc}
 & & & \langle AE\rangle & & & \\
 & \swarrow & & \downarrow & & \searrow & \\
\{ & - & & \langle AE\rangle & & \langle AE\rangle & \} \\
 & & \swarrow & \downarrow & \searrow & \downarrow & \\
 & & \Big\{+ & \langle \mathbf{3}\rangle & \langle \mathbf{4}\rangle\Big\} & \langle \mathbf{2}\rangle &
\end{array}$$

כעת נרצה לכתוב את האינטרפטר (לקבל את המשמעות), אבל נתחיל מלעשות $parsing$, והאתגר הוא שצריך להחזיק מבנה נתונים, שיודע לסנן את המידע הלא חשוב. בדוגמה למשל $3+4$ מבחינתנו יכול להיות גם $4+3$, ולמשל המידע שדוקא 3 ראשון, לא חיוני. ולכן נרצה לבנות: abstract syntax tree

המשך הדוגמה - בצורה אבסטרקטית נרצה ש

$$\begin{array}{ccc} & \langle Sub \rangle & \\ \swarrow & & \searrow \\ \langle Add \rangle & & \langle \boldsymbol{Num\ 2} \rangle \\ \swarrow \quad \searrow & & \\ \left\{ + \quad \langle \boldsymbol{Num\ 3} \rangle \quad \langle \boldsymbol{Num\ 4} \rangle \right\} & & \end{array}$$

וכעת לא משנה לנו האם:

- 3+4 (infix),

  `3 4 + (postfix),`

- +(3,4) (prefix with operands in parens),
- (+ 3 4) (parenthesized prefix),

נדגים ב $DrRacket$ בניה של מבנה נתונים כזה:

- במהלך התוכנית נעזר בפונקציה string->sexpr שמקבלת $string$ ומחזירה אובייקט $sexpr$ (כאשר הוא נתקל בסוגריים הוא מחשיב כתא אחד)
- נבנה טיפוס AE
- נבנה פונקציה רקרוסיבית שתפרסר ביטוי:

```
#lang pl
(define-type AE
    [Num Number]
    [Add AE AE]
    [Sub AE AE])
(: parse-sexpr : Sexpr -> AE)
(define (parse-sexpr expr)
    (cond
        [(number? expr) (Num expr)]
        [(and (list? expr) (= (length expr) 3) (equal? (first expr ) '+))
           (Add (parse-sexpr(second expr)) (parse-sexpr(third expr)))]
        [(and (list? expr) (= (length expr) 3) (equal? (first expr ) '-))
```

```
            (Sub (parse-sexpr(second expr)) (parse-sexpr(third expr)))]
        [else (error 'parse-sexpr "bad syntax in ~s" expr)]
))
(: parse : String -> AE)
(define (parse code)
    (parse-sexpr(string->sexpr code)))
(test (parse "5") => (Num 5))
(test (parse "{ + 3 4}") => (Add (Num 3) (Num 4)))
(test (parse "{- { + 3 4 } 5 }") => (Sub (Add (Num 3) (Num 4)) (Num 5)))
(test (parse "{+ 4 5 { - 4 5 4 5 4 5 }}") =error> "bad syntax")
result:
    4 tests passed.
```

0.3.2 פונקציית $match$

הקוד יצא לנו יחסית ארוך, ונרצה לקצר על ידי שימוש בפונקציה $match$, דוגמת שימוש:

```
(match var
    ['x "yes"]
    [else "no"])
```

דוגמה נוספת, בנוסף לבדיקת התאמה אני מחליט איך לקרוא למשתנים ומבצע את הפעולה שאני רוצה:

```
(match (list 1 2 3)
    [(list x y z) (+ x y z)]) ; evaluates to 6
(match '((1) (2) 3)
    [(list (list x) (list y) z) (+ x y z)]) ; evaluates to 6
match '((1 2) (3 4) (5 6) (7 8))
    [(list (list x y) ...) (append x y)])  ; evaluates to (1 3 5 7 2 4 6 8)
```

בדוגמה האחרונה x, y כ"א הן רשימה, והשלוש נקודות אומר לך להמשיך לשאר המערך , ובסוף נקבל שרשור של כל המערכים.
התנאי של ה $match$ יכול להיות אחד מ:

- **id** -- matches anything, binds 'id' to it
- _ -- matches anything, but does not bind
- (number: n) -- matches any number and binds it to 'n'
- (symbol: s) -- same for symbols (string: s) -- strings
- (sexpr: s) -- S-expressions (needed sometimes for Typed Racket)

- (and pat1 pat2) -- matches both patterns
- (or pat1 pat2) -- matches either pattern (careful with bindings)

וכעת נחליף בקוד שלנו את *cond* ב*match* ונקבל:

```
(: parse-sexpr : Sexpr -> AE)
(define (parse-sexpr expr)
(match expr
    [(number: n ) (Num n)]
    [ ( list '+ l r)
        (Add (parse-sexpr(l)) (parse-sexpr(r)))]
    [ ( list '- l r)
        (Sub (parse-sexpr(l)) (parse-sexpr(r)))]
    [else (error 'parse-sexpr "bad syntax in ~s" expr)]
))
```

כעת נרצה שהקוד שלך יעריך (=יחשב) את הביטויים , כלומר לברר את המשמעות שלהם:

```
#| eval's Foramlization
    eval(N) = N
    eval (+ E1 E2) = eval(E1) + eval(E2)
    eval (- E1 E2) = eval(E1) - eval(E2)
|#
(: eval : Sexpr -> Number)
    (define (eval expr)
    (match expr
        [(number: n ) n]
        [(list '+ l r) (+ (eval l) (eval r))]
        [(list '- l r) (- (eval l) (eval r))]
        [else (error 'eval "bad syntax in ~s" expr)]
    ))
(: run : String -> Number)
(define (run code)
    (eval (string->sexpr code)))
(test (run "5") => 5 )
(test (run "{ + 3 4}") => 7)
(test (run "{- { + 3 4 } 5 }") => 2)
(test (run "{+ 4 5 { - 4 5 4 5 4 5 }}") =error> "bad syntax")
```

result:

```
    4 tests passed.
```

מה החיסרון של הקוד הזה?

- שיש לנו ערבוב בין הפעולות הנדרשות: להבין את הביטוי ולחשב אותו, ואנחנו נעדיף להפריד בינהם.
- איך זה יעזור?
  - לדוגמה מצאנו דרך יעילה יותר לחשב, לא צריך לשנות את כל הקוד, אלא רק את החלק שמחשב
  - לדוגמה מחליטים לשנות את מבנה הביטויים, אז משנים רק אותו והחישוב עומד בפני עצמו
- בשיעור הבא נראה מימוש נכון של $eval$.

הרצאה 4

תזכורות

<u>דוגמה ל $Ambiguity$:</u>

eval(1 - 2 + 3) = eval(1 - 2) + eval(3) [b] = eval(1) - eval(2) + eval(3) [c] = 1 - 2 + 3 [a,a,a] = 2

eval(1 - 2 + 3) = eval(1) - eval(2 + 3) [c] = eval(1) - (eval(2) + eval(3)) [a] = 1 - (2 + 3) [a,a,a] = -4

וברור שאנחנו לא יכולים להרשות לעצמנו בדקדוק שאנחנו בונים, פתרנו זאת על ידי סוגריים.

0.3.3 Compositionality

תכונה חשובה נוספת היא Compositionality

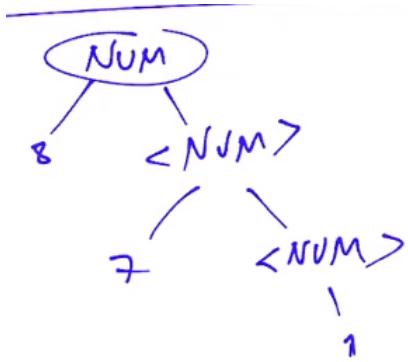
נסביר זאת דרך דוגמה והבעיה הבא:

נניח שיש לנו דקדוק:

<NUM> ::= <digit> <NUM>|<digit>

<digit> ::= 0|1|....|9

למשל המספר 871 ייוצג כך:



כעת נניח שהיינו רוצים לכתוב פונקציה שמחזירה את הערך של הביטוי, לכן יתקיים ש:

eval(0)=0,eval(1)=1 .....

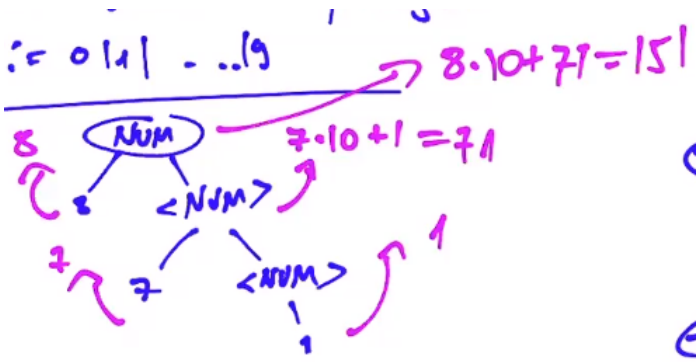
וכנראה שאם היינו מגדירים מספר כללי היינו מנסים:

$$\text{eval}(\langle digit\rangle\,\langle Num\rangle) = eval\,(\langle digit\rangle) + eval\,(\langle Num\rangle)$$

הבעיה שהמחרוזת 17 היא לא הסכום של $1+7$ ובשביל לפתור את זה, צריך את הנוסחה הבאה:

$$eval\,(17) = 10 * eval\,(1) + eval\,(7)$$

נשים לב שאם נגדיר את הנוסחה הזו בצורה רקורסיבית עדיין יש לנו בעיה, נראה זאת בעזרת העץ:



וברור ש $eval\,(871) \neq 151$ לכן צריך לשפר גם את הנוסחה הזו.

מסקנה נשכלל את הנוסחה, ויתקיים ש:

$$eval\,(871) = 10^2 * 8 + 10^1 * 7 + 10^0 * 1 = 871$$

השיפור שהחזקה תהיה לפי גובה העץ, הבעיה שחישוב כזה הוא "מורכב" , כלומר דורש **חישובים התלויים בעץ כולו** ואנחנו רוצים שהחישוב יהיה מקומי כלומר שNum ידע לחשב זאת ללא תלות בעץ ואם נצליח נקיים את ה Compositionality.

הגדרה פורמלית Compositionality:

$eval$ על קודקוד T עם בנים $T_1, ..T_k$ יבצע :

א. $v_1 = eval\,(T_1)\,, v_2 = eval\,(T_2)\,....v_k = eval\,(T_k)$

ב. יבצע פעולה מקומית על $v_1, ..., v_k$

עכשיו אם נחזור לדוגמה שלנו, נוכל לעשות שיפוץ קל ולהרויח את התכונה הנ"ל (נחליף מיקום של $digit$⇔ העץ משוך שמאלה:

<NUM> ::= <NUM><digit> | <digit>|

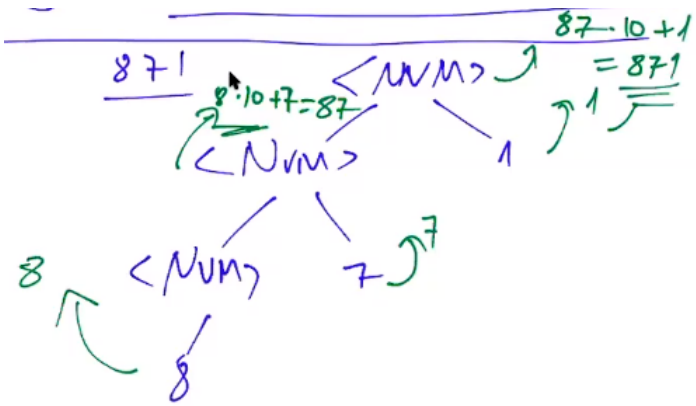
<digit> ::= 0|1|....|9

ונקבל ש:

$\text{eval}(\langle digit\rangle) = \langle digit\rangle$

$\text{eval}(\langle Num\rangle\,\langle digit\rangle) = eval\,(Num) + 10 * eval\,(\langle digit\rangle)$

נשים לב שכל הפעולות הן מקומיות $(eval, *10)$ והעץ יראה כך:



ואכן $eval\,(871) = 871$, כנדרש.

<u>תזכורת קצרה אחרונה</u>, בשיעור שעבר הגענו לקוד הבא:

```
(: eval : Sexpr -> Number)

    (define (eval expr)
    (match expr

        [(number: n ) n]
        [(list '+ l r) (+ (eval l) (eval r))]
        [(list '- l r) (- (eval l) (eval r))]
        [else (error 'eval "bad syntax in ~s" expr)]

    ))
```

ואמרנו שהקוד הזה עובד נהדר, אבל יש לו בעיה שה$syntax$ וה $semantics$ מעורבים זה בזה , ואנחו רוצים להפריד, שיהיה חלק שיעשה $parsing$ וחלק שיעשה את ה $evaluator$, והסברנו בשיעור שעבר למה אנחנו מעוניינים בחלוקה הזו

במילים אחרות, נרצה שהטסטים הבאים יעברו:

```
(test (eval (parse "3"))              => 3)
(test (eval (parse "{+ 3 4}"))        => 7)
(test (eval (parse "{+ {- 3 4} 7}")) => 6)
```

כלומר פונקציה $parse$ שעוברת על הביטוי סינטקטית ומאשרת אותו, ואת הביטוי הזו הפונקציה $eval$ מעריכה ומחשבת.

נכתוב את כל הקוד מהתחלה:

```
#lang pl
(define-type AE

    [Num Number]
    [Add AE AE]
    [Sub AE AE])

(: parse : String -> AE)
(define (parse code)
```

```
    (parse-sexpr (string->sexpr code)))

(: parse-sexpr : Sexpr -> AE)

    (define (parse-sexpr sexpr)

        (match sexpr
        [(number: n ) (Num n)]
        [(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]
        [(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]))

(: 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))]))

(: run : String -> Number)

    (define (run code)
    (eval (parse code)))

(test (run "3") => 3)
(test (run "{ + 3 4 }") => 7)
(test (run "{ - { + 3 4 } 6 }") => 1)
```

שאלה האם מתקיימת תכונת ה Compositionality?

כן, בגלל ש:

- במקרי הבסיס אנחנו נותנים מיידית, ערך ל'עלה'
- במקרי הריקורסיה אנחנו מבצעים חיבור/חיסור בין טיפוסים שקיבלנו

נעיר שבשפות תכנות תמיד נרצה את התכונה Compositionality , אבל לעולם לא נצליח להשיג במלואה, כיון שיש לנו שמות מזהים (=משתנים) והערך של משתנה מצריך מאיתנו את ה"תמונה הגדולה".

שיעור 5

בשיעור שעבר פיתחנו שפה, שעובדת ורצה. מספר הערות:

- ניקח לדוגמה את הביטוי : "{* {+ 2 4}{+ 2 4}}" הסוגריים המסולסות "נעלמות" כאשר אנחנו מפעילים את הפונקציה string->sexprשיוצרת לנו את הרישמה: ((4 2 +) (4 2 +) *)
- נשים לב לכפילות של "{ + 2 4 } " ונרצה לצור שמות־מזהים (=משתנים) כלומר היינו רוצים x = {+ 2 4 }, למה?
  - קוד קריא יותר
  - יעילות - (לחשב פעם אחת)
  - להמנע משכפול קוד: קל לתחזק ומונע באגים
  - כח ביטוי למתכנת, למשל הביטוי{salary salary * } נותן את האפשרות להבין:
    - שדווקא לקחתי את אותו ערך
    - שהערך קשור לעולם המשכורות

0.3.4 WAE ו $With$

Let : נזכיר את המבנה של let ב $reacket$: (let ([x (+ 4 2) (* x x)).

נרצה לצור משהו דומה אצלנו, נבחר את המילה $with$, ונרצה לאפשר זאת בשפה שלנו. לכן:

- נשפר את מבנה הנתונים, ונקרא לו WAE
- ונגדיר שאנחנו רוצים שיתקיים:

```
<WAE> ::=

    <num>
    | { + <WAE> <WAE> }
    | { - <WAE> <WAE> }
    | { * <WAE> <WAE> }
    | { / <WAE> <WAE> }
    | { with { <id> <WAE> } <WAE> }  ;;; {with {name named-expression } body }
    | <id>
```

- while :
  - <num> : identifies any expression that pl evaluates to a Number
  - <id> : : identifies any expression that pl evaluates to a Symbol

- הערה : מבחינה תחברית ביטוי כמו "x","y" הוא תקין , כי כעת יש את הכלל <id>→ <WEA> אבל מבחינת סמנטית הם לא (במקרה ורשמנו רק "x" ללא הצהרה.
- נרצה שהמתשמש ידע אם עשה טעות רגילה או טעות שקשורה ל $with$, ולכן נפריד למקרים.

```
(define-type WAE

    [Num Number]
    [Add WAE WAE]
    [Sub WAE WAE]
    [Mul WAE WAE]
    [Div WAE WAE]
    [Id Symbol]
    [With Symbol WAE WAE])

(: parse : String -> WAE)

    (define (parse code)
    (parse-sexpr (string->sexpr code)))

(: parse-sexpr : Sexpr -> WAE)
(define (parse-sexpr sexpr)
(match sexpr
```

```
    [(number: n ) (Num n)]
    [(symbol: name) (Id name)]
    [(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]
    [(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]
    [(list '* l r) (Mul (parse-sexpr l) (parse-sexpr r))]
    [(list '/ l r) (Div (parse-sexpr l) (parse-sexpr r))]
    [(cons 'with _ )
        (match sexpr
        [(list 'with (list (symbol: name) named-expr) body)
           (With name (parse-sexpr named-expr) (parse-sexpr body))]
        [else (error 'parse-sexpr "bad 'with' syntax in ~s" sexpr)])]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(test (parse "5") => (Num 5))
(test (parse "{ + 3 4 }") => (Add (Num 3) (Num 4)))
(test (parse "x") => (Id 'x)) (test (parse "{ + 4 2 }") => (Add (Num 4) (Num 2)))
(test (parse "{ * x x }") => (Mul (Id 'x) (Id 'x)))
(test (parse " { with {x { + 4 2 }} {* x x }}") => (With 'x(Add (Num 4) (Num 2))
(test (parse "{ with x {+ 4 2 } {* x x }}") =error> "bad 'with'")
```

- $eval$ ל $with$, מצריך את השלבים הבאים:
  1. הערך את name-expr וקבל ערך v
  2. החלף את המופעים המתאימים של $name$ בערך v בתוך $body$
     (א) הסינטקס: [name/v]body
  3. הערך את $body$
  - בדוגמה שלנו תחשב $eval\,(+\ 4\ 2)$ תכניס את $x \leftarrow 6$, ותבצע {* 6 6}
- לכן צריך לדעת לעשות החלפה $Subst =$. ננסה להגדיר פורמלית את $[x/v]\,e$
  - <u>נסיון 1:</u> החלף את כל המופעים של x בערך v :

$$\{\text{with } \{\overbrace{x}^{x}\ \overbrace{x}^{v}\}\overbrace{\{+\ \text{x x}\}}^{e} \Rightarrow \{+\ 5\ 5\}\checkmark$$

$$\{\text{with } \{\text{x } 5\ \}\{+\ 10\ 4\} \Rightarrow \{+\ 10\ 4\}\ \checkmark$$

הבעיה:

$$\{\text{with } \{\text{x } 5\ \}\{+\text{x } \{\text{with } \{\ \text{x } 3\}\ 8\ \} \Rightarrow \{\{+\ 5\ \{\text{with } \{\ 5\ 3\}\ 8\ \}\textbf{bad syntax}$$

ובשביל לפתור אותו נגדיר מושגים חדשים, שמטרתם להבדיל בין הx הפנימי לחיצוני:

* Binding instance - מופע של x , שבו אני מקשר את x לביטוי כלשהו (הצהרה על שם מזהה x)
* Scope - עבור Binding instance אותו חלק בקוד שבו כל מופע של x מקושר למופע ההצהרתי הזה

* Bound instance - כל מופע של x שאינו Binding instance וגם נמצא בתוך $with$ של Binding instance
* Free instance - כל מופע של x שאינו Binding instance וגם אינו Bound

לדוגמה:

$$\text{e} = \{\text{with } \{ \underbrace{x}_{Binding} \ 5 \}\{+\text{x } \{\text{with } \{ \underbrace{\text{x x}}_{Bound} \}\}$$

– <u>נסיון שני:</u> החלף את כל המופעים של x בתוך e שאינם $binding$ בערך v

$$\{\text{with } \{\text{x } 5 \}\overbrace{\{+\text{x } \{\text{with } \{ \text{x } 3\} \ \mathbf{10} \}\}}^{e} \Rightarrow \{\{+ \ 5 \ \{\text{with } \{ \text{x } 3\} \ 10 \}\checkmark$$

$$\{\text{with } \{\text{x } 5 \}\overbrace{\{+\text{x } \{\text{with } \{ \text{x } 3\} \ \mathbf{x} \}\}}^{e} \Rightarrow \{\{+ \ 5 \ \{\text{with } \{ \text{x } 3\} \ \mathbf{5} \}\chi \ \mathbf{x \text{ should be } 3}$$

– <u>נסיון שלישי:</u> החלף את כל המופעים של x בתוך e שאינם $binding$ וגם אינם בתוך ה $scope$ של מזהה עם אותו שם.

$$\{\text{with } \{\text{x } 5 \}\{+ \text{ x x}\}\} \Rightarrow \{+ \ 5 \ 5 \} \ \checkmark$$

$$\{\text{with } \{\text{x } 5 \}\overbrace{\{+\text{x } \{\text{with } \{ \text{x } 3\} \ \mathbf{x} \}\}}^{e} \Rightarrow \{\{+ \ 5 \ \{\text{with } \{ \text{x } 3\} \ \text{x} \}\checkmark$$

$$\{\text{with } \{\text{x } 5 \}\overbrace{\{+\text{x } \{\text{with } \{ \text{y } 3\} \ \mathbf{x} \}\}}^{e} \Rightarrow \{\{+ \ 5 \ \{\text{with } \{ \text{y } 3\} \ 5 \}\checkmark$$

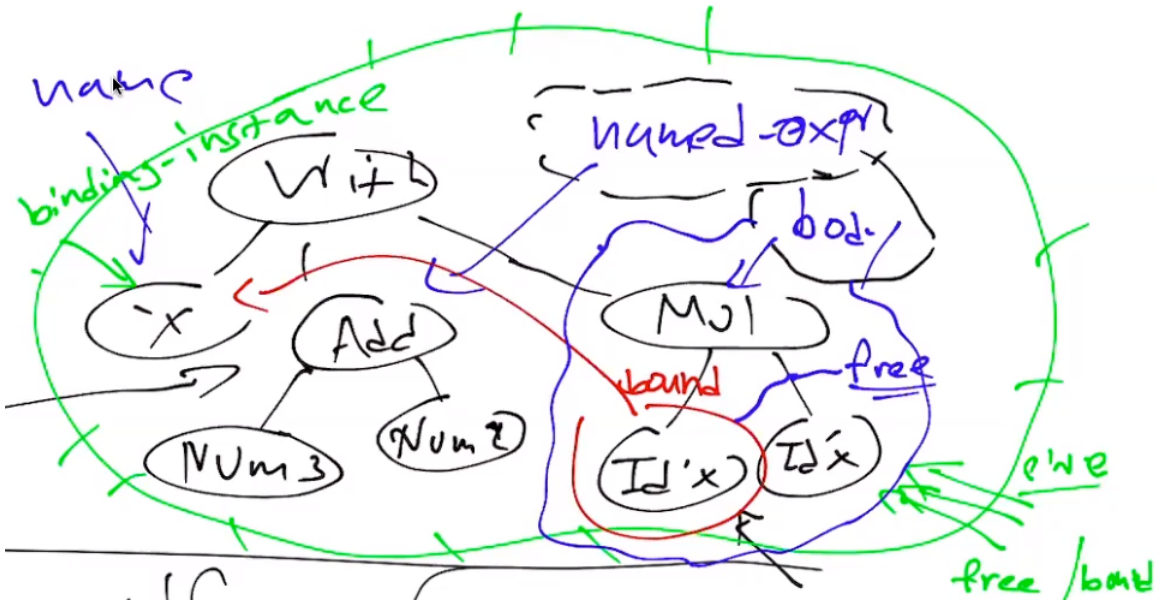
- לסיכום: $[x/v]e$ - החלף את כל המופעים החופשיים של x בתוך e
- שיעור הבא נשלים את קטע הקוד החסר

שיעור 6

חזרה על $with$ - בשיעור שעבר דיברנו על ה $parsing$ וכעת נרצה לבצע לו $eval$, נזכיר את השלבים:

1. val $\leftarrow$ (eval named-expr)
2. subst body name **val** $\equiv [name/val]\, body \equiv [x/v]\, e$
   - הגדרנו $[x/v]\, e$: החלף את כל <u>המופעים החופשיים</u> של x בתוך e בערך v
3. eval body

נזכיר גם את המושגים: $binding - instance\,(green)\,, free\,(purple)\,/bound\,(red)$



היום:

1. נכתוב פרוצדורה $subst$ - עושה פעולת תחבירית. מקבלת ערך חדש שם ביטוי, מחליפה את כל המופעים החופשיים של שם בתוך ביטוי בערך החדש.
2. נכתוב את $eval$ (מחשב) בעזרת $subst$

לדוגמה הביטוי:

```
(eval ( with 'x add (Num 2) (Num 3))
            from
    (Mul (Id 'x) (Id 'x)))) // := body
```

steps:

```
1. eval (Add (Num 2) (Num 3)) // := to
2. subst := body from to →
3. eval (Mul (Num 5 ) (Num 5))
```

הערות:

1. נשים לב שבמקרה של Id מראש לא נקבל משתנה חופשי כי נמנע מהריקורסיה להכנס למקרה הזה ב $\boldsymbol{With}$
2. נשים לב שב$test$ הבא:

```
(test (subst
   (With 'x ;; name
      (Num 3) ;; name-expr
      (Add (Id 'x) (Num 5 ))) ;;body . note: 'x is not free var
   'x ;; from
   (Num 8)) ;; to
   => (With 'x
```

```
      (Num 3)
      (Add (Id 'x) (Num 5))))
```

ה x' הוא **לא משתנה חופשי** ולכן נשאיר אותו כמות שהוא, לעומת זאת כאן את x' ב $(Num\ 8)$

```
(test (subst
   (With 'y ;; name
      (Num 3) ;; name-expr
      (Add (Id 'x) (Num 5 ))) ;;body . note: 'x is bound var
   'x ;; from
   (Num 8)) ;; to
=> (With 'y
     (Num 3)
     (Add (Num  8) (Num 5))))
```

3. נשים לב שגם את $named-expr$ צריך לשלוח ל$subst$

4. אם $name = from$ נרצה להחזיר את ה $body$ כמו שהוא בלי החלפה

בסה"כ:

```
(: subst : WAE Symbol WAE -> WAE )
;; takes a WAE trre expr and a name and a value
;; and return a new tree with the same  structure
;; but without aby free instances of name (these are replaced by value
(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 name named-expr body)
       (With name ; name
       (subst named-expr from to) ; substituted name-expr
       (if (eq? name from)
          body ; non-substituted body
          (subst body from to)))])) ;; substituted
```

כעת נכתוב את $eval$ לפי השלבים שהזכרנו בתחילת השיעור:

```
#| How to evaluate (With name named-expr body)
1. v <- (eval named-expr)
2. body <- (subst body name v)
```

```
3. (eval body)
|#
(: 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))]
    [(Id name) (error 'eval "free identifier ~s" name)]
    [(With name named-expr body)

        (eval (subst body name (Num (eval named-expr))))]))

(: run : String -> Number)
(define (run code)

    (eval (parse code)))

;;----------------- tests --------------------
(test (run "5") => 5)
(test (run "z") =error> "free ")
(test (run "{+ 5 5}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}}

    {with {y {- x 3}}
    {+ x y}}}") => 17)

(test (run "{with {x 5}

    {+ x {with {x 3} 10}}}") => 15)

(test (run "{with {x 5}

    {+ x {with {x 3} {+ x x}}}}") => 11)

(test (run "{with {x {+ 3 4}} {+ x {with {y 3} x}}}") => 14)
(test (run "{with {x 5}

    {with {y {* x 3}}
    {- y x}}}") => 10)

(test (run "{with {x 8}

    {with {x {* x x}}
    {/ x 4}}}") => 16)

(test (run "{with {x 1} y}") =error> "free identifier")
```

שיעור 7

0.3.5 $FLANG$ ופונקציות

תזכורת: בשיעור שעבר בנינו $interpeter$ שיודע לחשב ביטויים ארימתטים, נרצה היום לשפר אותו:

1. נרצה לפתח מנגנון שיודע להגדיר פונקציה אנונומית
   - לכן נניח והגדרנו את:

```
{fun {x}
   {* x x}}
```

2. נרצה לפתח מנגנון לקריאה ($call$) של לפונקציה = איך נפעיל אותה?
   - נשתמש בתבנית הרגילה. לדוגמה תהיה פונקציה f כלשהי אז ב $Racket$ נפעיל אותה כך:

```
(f 5)
```

   - אילו רצו להיות מפורשים יותר היו מוסיפים $call$:

```
(call f 5)
```

   - אצלנו:

```
{call f 5}
{call {fun {x}
  {* x x}} 5}
```

- נשים לב שהצורה הזו לא "נוחה", וגם לא נותנת לנו את הכח של הפונקציות , ולכן נרצה להגדיר פונקציה ולהשתמש בה שוב ושב
  - הערה: להגדיר פונקציה נותן לי את האפשרות לשמור את אופן הפעולה (את ה'איך') , ולחזור עליה שוב ושב :

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

ישנן 3 גישות לסוגי הפונקציות:

1. בעבר פוקציות היו אובייקטים שיודעים לבצע "חישוב" על נתונים מסוים (first order)
2. כיום מבינים שזהו אובייקט הרבה יותר מורכב, וניתן להביע איתו חישובים ותהליכים ברמה יותר גבוהה(high order) , ויותר קריאה.
   (א) ומכאן מגיעות האפשרויות של שליחת וקבלה פונקציה כמשתנה
3. שפות בהן פונקציה היא טיפוס כמו כל טיפוס אחר (First class) $\Leftarrow$ נותן לנו את האפשרות לפונקציה ללא שם (אנונימית)
   (א) מאפשר להגדיר פונקציה בזמן ריצה
   (ב) יכולות להשמר במבנה נתונים, ועוד.

נדגים את היתרון בשפה עילית: להלן קוד בשפה תחתית (first order) , מה הוא מבצע?

```
x = b * b
y = 4 * a
y = y * c
x = x - y
x = sqrt(x)
y = -b
x = y + x
y = 2 * a
s = x / y
```

תשובה, קשה לומר. לעומת זאת קוד בשפה עילית (First class) :

```
(-b + sqrt(b^2 - 4*a*c)) / 2a
```

"מספר" לנו שזה פתרון למשוואה ריבועית, ובכך הרווחנו יכולת הבעה. הרעיון מאחורי זה שפונקציה היא טיפוס בפני עצמו, ולא צריך הגדרה (כמו ש5 לא צריך הגדרה)

דוגמה נוספת מjs :

```
function foo(x) {
    function bar(y) { return x + y; }
    return bar;
}
function main() {
    var f = foo(1);
    var g = foo(10);
    alert(">> "+ f(2) + ", " + g(2));
}
```

בתוכנית שלנו, יש

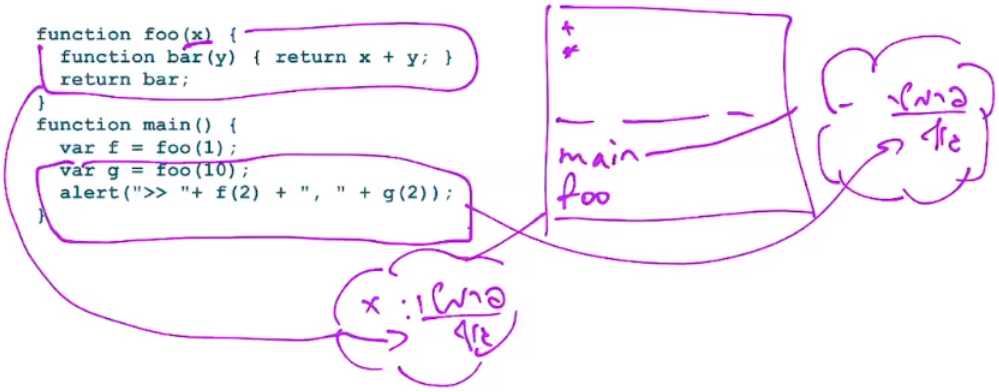
- פעולות רגילות $+, -$
- פונקציות:

$main$ –

foo –

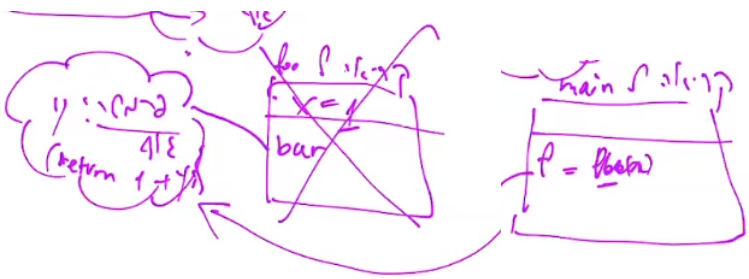
נרצה לחשוב על הפונקציות כענן , כלומר משהו שאנחנו לא מחשבים כרגע, ויום אחד נשתמש בו.

הענן הוא קופסה סגורה שיודעת לקבל פרמטר, ויש לה גוף (פעולות החישוב שלה)

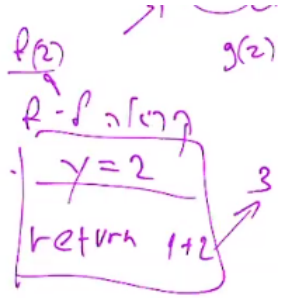


<u>נרוץ עם התכנית:</u>

- קריאה ל $main$ - ללא פרמטר
- קריאה לfoo עם פרמטר $x = 1$
  - גוף : bar
- קריאה לbar : עם פרמטר y
  - גוף return x+y=1+y כי x משתנה חופשי
- ולכן $f = y + 1$



- כנ"ל ל g
- $f(2)$ (נזכר שf **מחזיק פונקציה**) שהפרמטר שלה 2 , הגוף return 1+2



- כנ"ל ל $g = 10 + 2 = 12$
- סה"כ: $3, 12$

לסיכום:

- בהדגמה הראינו איך הפונקציה "נוצרת בזמן ריצה"
- נתינת השם bar לפונקציה , די מיותר והיה אפשר להשאר ללא השם
- אותו קוד ב$racket$:

```
(define (foo x)
   (define (bar y) (+ x y)) bar)
```

נגדיר את ה BNF שלנו - נוסיף את $fun, call$

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

נשים לב שהגדרנו בfun את האפשרות לקריאה כזו:

```
(7 6)
```

כלומר מספר שקיבל שם של פונקציה. **אבל** זו בעיה **סמנטית,** ותחבירית זה תקין ($racket$ עצמה אפשרו זאת, בpl חסמו)

הקוד ל$parse$:

```
(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] )
(: parse : String -> FLANG)
    (define (parse code)
(parse-sexpr (string->sexpr code)))
(: parse-sexpr : Sexpr -> FLANG)
    (define (parse-sexpr sexpr)
    (match sexpr
```

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

שיעור 8

בשיעור שעבר , הגדרנו פונקציות וממשנו את החלק של ה $parse$

היום נרצה לממש את $eval$

נתחיל מדוגמאות :

- דוגמה בסיסית - לביטוי:

```
{ fun {x} {+ x 1}}
```

  - נרצה שיוחזר משהו בסגנון:

```
[param: 'x
body: (Add (Id 'x) (Num 1))]
```

- דוגמה נוספת:

```
{ with { x 1 }
    {fun {y} {+x y}}}
```

  - אז מכיון ש $x = 1$ (משתנה) נקבל:

```
[param: 'y
body: (Add (Num 1) (Id 'y))]
```

0.4 $subst$ ומודל ההחלפות

לכן נרצה להגדיר באופן פורמלי את $subst$ שלנו

```
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}
{call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}
{fun {y} E}[v/x]      = {fun {y} E[v/x]}
{fun {x} E}[v/x]      = {fun {x} E }
```

הערות:

- הביטוי:

```
{fun {y} E}[v/x]      = {fun {y} E[v/x]}
```

  - משמעו תחליף ב E את הכל הx , וזה לא משפיע על הפונקציה שלנו, כי הפרמטר שלה הוא y

- לעומת זאת הביטוי

```
{fun {x} E}[v/x]      = {fun {x} E }
```

  - מחייב אותנו **לא** להחליף כי הפונקציה כולו תלויה בערך שx יתקבל כפרמטר, ולכן הוא לא משתנה חופשי

- אבל כאשר קוראים לפונקציה:

```
{call E1 E2}[v/x]     = {call E1[v/x] E2[v/x]}
```

  - נרצה להחליף כי זה כבר השימוש בפונקציה

קוד ל $subst$:

```
(: 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 name named-expr body)
```

```
        (With name ; name (subst named-expr from to) ; substituted name-expr
           (if (eq? name from)
             body ; non-substituted body
             (subst body from to)))] ;; substituted
     [(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 נעבור ל $eval$:

- מטרתנו ש $eval$ תחזיר שני סוגים $(types)$:
  - $Number$
  - $functions$ (ענן בסגנון):

```
[param: Symbol
body: (AST FLANG]
```

  - דוגמאות:

```
(eval (Num 1))
(eval (Fun 'x (Add (Id 'x) (Num 1))))
```

- לכן נשנה את הגדרת $eval$ ל $FLANG$

  - נציין שזהו פתרון בעייתי , כי האובייקט $FLANG$ לכשעצמו מכיל בנאים שלא קשורים, והוא תוצר של תהליך $parse$ וזה מוזר לערבב אבל בכל מקרה נבצע תיקון מעין זה בהמשך, ולכן כעת נשאר עם הפתרון הזה.
  - נרצה שידע להתמודד עם הטסטים הבאים:

```
(test (eval (Num 1 )) => (Num 1))
(test (eval (Mul (Num 1) (Num 6))) => (Num 6))
(test (eval (Fun 'x (Add (Id 'x) (Num 1)))) =>
            (Fun 'x (Add (Id 'x) (Num 1))))
(test (eval (With 'y
                  (Num 5)
                  (Fun 'x (Add (Id 'x) (Num 'y))))) =>
                  (Fun 'x (Add (Id 'x) (Num 5))))
```

נגדיר פורמלית את $eval$:

```
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(FUN)       = FUN ; assuming FUN is a function expression
eval({call E1 E2})=  if {fun {x} Ef } <--- eval (E1)
                     then  eval(Ef[x/eval(E2)])
                     else Error
eval(Ef[eval(E2)/x])   if eval(E1) = {fun {x} Ef}
                  = error!                 otherwise
```

הערות:

- id יתן $error$, כי אין משמעות ל "x"
- כפי שאמרנו Fun נחזיר כמו שהוא, כי מחכה לפרמטר בשביל לקבל משמעות
- ב $call$ בif , נרצה לחשב את $eval(E2)$ בשביל יעילות, שהפונקציה כבר תקבל ערך מחושב, ולא תצרך ללכת ולחשב אותו, לדוגמה:

```
{call {Fun {x} { + x x }}}
   {* 5 { * 17 {* 2 3}}}}
```

  - אז ניתן לחשב את x בכל פעם או מראש לחשב (וזה מה שנעדיף) ושהפונקציה תעבוד עם 510

- נשים לב שבגלל ששינו את הערך המוחזר ל$FLANG$ אנחנו צריכים לבצע התאמות בכל מקום שהפונקציה מחזירה מספר למשל (Add (Num 1) (Num 2))

  - לכן צריך לכתוב פונקצית עזר arith-op שתדע לבצע את הפעולות הארימתטיות בין המספרים ולהחזיר $FLANG$ מתאים

```
{call {With {y 2}
     {fun {x} {+ x y}}}
   3}
```

- כחלק מהשינוי ל$FLANG$ ב $eval$, נרצה בrun להחזיר את המספרי ולכן נוסיף בדיקה שתעזור לנו.

הקוד:

```
(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)
;; gets a Racket numeric binary operator, and uses it within a FLANG
;; 'Num' wrapper (note H.0 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))))


(: 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-e arg-e) ; <- nested pattern
           (let ([funv (eval fun-e)])
             (cases funv
               [(Fun name body)
                 (eval (subst  body name (eval arg-e)))]
               [ else (error 'eval "expected a function ,got ~s" funv)]))]))
(: 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)])))
```

שיעור 9

תזכורת:

- כתבנו את האובייקט $FLANG$
- $Subst$
- $Eval$ -
- כתבנו גם את $Call$ ואת $With$

0.5 מודל ה SubstCache

נטען שהביטויים הבאים זהים:

```
(Call (Fun name body)
    E)
(With name
    E
    body)
```

דוגמאות לדמיון:

```
{Call {fun {x} {* x 7}}
    5}
{With {x 5} {* x 7}}
```

אם כן למה המצאנו את שני המגנונים $Call, With$? או מה היחס בין $Call$ ל $With$?

- מצד אחד הביטוי $with$ הרבה יותר קריא.
  - למעשה $With$ הוא syntactic sugar של $Call$, והוא מקרה פרטי שלו (ב$Racket$ ה let הוא באמת syntactic sugar לקריאה לפונקציות)
- מצד שני, היחוד של $Call$ הוא לפונקציות שאנחנו מגדירים ,ולא משתמשים באותו רגע. למשל:

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

  - היכולת להגדיר את sqr , ואז לקרוא לה פעמיים היא בזכות הגדרת ה $call$

מה למדנו כאן?

- שבביטוי ({call E1 E2 }eval, $E1$ הוא לא בהכרח פונקציה ברגע הקריאה, ויכול להיות שרק לאחר ההערכה שלו ($eval$) באמת נקבל צורה של ..Call..Fun פורמלית:

```
if eval (E1)={fun {x} Ef}
   then eval (Ef[eval/x])
else    error!
```

דוגמה הרצה של $with, call$

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

הראנו ש $With$ הוא בעצם כמו $Call$ $\Leftarrow$ ניתן להתיחס ל$with$ כמו לפונקציה.

החסרון ש$with/call$ כל קריאה לפונקציה גורמת לנו לחזור על הקוד שוב ושוב בשביל לבצע את ההחלפות, וזה יוצר חוסר יעילות מאוד גדול. לכן נרצה ל"שמור" במעבר ראשון את כל הפונקציות , בשביל שבמעבר שני כבר נוכל לבצע את כל ההחלפות ולא לסרוק את הקוד שוב ושוב. לשם כך, נממש מעין מחסנית של פונקציות. נקרא לו:

```
;; a type for substitution caches:
(define-type SubstCache = (Listof (List Symbol FLANG)))
;;
(: empty-subst : SubstCache)
(define empty-subst null)
;;
(: extend : Symbol FLANG SubstCache -> SubstCache)
(define (extend id expr sc)
  (cons (list id expr) sc))
;;
(: 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))]))
```

שיפור: נשתמש בפונקציה של $racket$ שיודעת לעבוד עם מבנה של רשימה של רשימות בשם $assq$, לדגומה:

```
> (assq 3 (list (list 1 2) (list 3 4) (list 5 6)))
'(3 4)
> (assq 7 (list (list 1 2) (list 3 4) (list 5 6)))
#f
```

ולכן נוכל להשתמש בה (הכי קרוב למילון ב $Racket$)

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

ב$parser$ אין מה לשנות, ב$eval$ צריך לשנות:

- $eval$ צריכה עכשיו לקבל את המחסנית, ולעבור בין כל הקריאות הרקורסיביות
- $eval(id)$ - ידרוש ממנו ללכת לחפש במחסנית (look-up) אם זו פנוקציה מוכרת,
  - $lookup$ צריכה לדעת להחזיר רק את המתשנה שאני מחפש.
- $eval(with)$ - יגרום לנו להוסיף לתחילת המחסנית משתנה עם השם והערך

– נשים לב שאם יש לנו $with$ בתוך $with$ עם אותו שם למשתנה (למשל x) אז המשתנה החיצוני ידרס, וזה לא פוגע במשמעות כי ההחלפות מתבצעות בסדר של שכבות הריקורסיה

- $eval(call)$ דומה ל$with$ ועל פי הלוגיקה שהסברנו בתחילת השיעור.
- **נשים לב:** ניתן למחוק כעת את $Subst$

Lookup rules:

lookup(x,empty-subst) = error!

lookup(x,extend(x,E,sc)) = E

lookup(x,extend(y,E,sc)) = lookup(x,sc) if 'x' is not 'y'

Evaluation rules:

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

השינויים בקוד עצמו ל$eval$:

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

והשינויים בrun :

- צריך להוסיף את המחסנית (sc) כרשימה ריקה ולשלוח אותה ל $eval$

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

: $test$

```
(test (run "{with {x 3}
    {with {f {fun {y} {+ x y}}}
        {with {x 5}
            {call f 4}}}}")
```

- החלפה ראשונה:

```
(test (run "{with {x 3}
   {with {f {fun {y} {+ 3 y}}}
      {with {x 5}
        {call f 4}}}}")
```

- החלפה שניה:

```
(test (run "{with {x 3}
   {with {f {fun {y} {+ x y}}}
```

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

- החלפה שלישית:

```
(test (run "{with {x 3}
    {with {f {fun {y} {+ x y}}}
       {with {x 5}
         {call {fun {y} {+ 3 4}} 4}}}}")
```

– לכאורה נקבל $y = 3 + 4 = 7$, אבל ב$racket$ אנחנו מקבלים 9 , האם זה נכון?
תלוי במודל, ישנם שני מודלים דינאמי וסטטי ונרחיב עליהם בשיעור הבא.

שיעור 10

0.5.1 המודל הדינמי VS המודל הסטטי

בשיעור שעבר הגדרנו מעין מבנה נתונים SubstCache, שמשפר לנו את העבודה עם פונקציות.

והזכרנו את הטסט הבא, כטסט בעייתי:

```
 (test (run "{with {y 4}
     {with {foo {fun {x} {* x y}}}
        {with {y 6}
           {call foo 7}}}}")
```

נריץ את הטסט הזה, על פי מודל ההחלפות:

1. החלפה ראשונה - {with {y 4}..} - דורש את חישוב ה$body$ לאחר החלפת כל המשתני y החופשיים ל 4 , (הy השני אינו חופשי) לכן:

```
test (run "{with {y 4}
    {with {foo {fun {x} {* x 4}}}
       {with {y 6}
         {call foo 7}}}}")
```

2. החלפת $with$ שניה לfoo :

```
test (run "{with {y 4}
    {with {foo {fun {x} {* x 4}}}
       {with {y 6}
         {call {foo {fun {x} {* x 4}} 7}}}}")
```

3. עכשיו צריך לחשב את ה $body$, אבל יש לנו שוב {with {y 6}...}, ולכן צריך להחליף את כל המשתנים החופשיים - אין לנו. אז נתחל לחשב את ה$body$:

```
{call {foo {fun {x} {* x 4}} 7}
```

התוצאה 28

עכשיו נריץ לפי שלבי הריצה של מודל ה SubstCache:

- בשלב ראשון נתחיל עם sc=[]
- בשלב שני נקבל : sc=[(y 4)]
- בשלב שלישי נקבל: sc=[(foo {fun {x {* x y}}) (y 4)]
- בשלב רביעי נקבל: sc=[(y 6) (foo {fun {x {* x y}}) (y 4)]
- בשלב חמישי נקבל: call foo ולכן נקבל את את {fun {x {* x y}
- בשלב השישי נגיע לחישוב ה*body* של הפונקציה , כלומר לחשב את {* x y} על ידי האיבר $x = 7$
  - בשלב זה המחסנית נראית כך: sc=[(x 7) (y 6) (foo {fun {x {* x y}}) (y 4)]
  - לכן *lookup* לx יחיזר 7, ול y יחזיר 6 ,

  והתוצאה תהיה 42 $\rightarrow$ {* 6 7 }

כפי שראינו, שתי הריצות הגיוניות ונותנות תוצאות שונות. האם זה באג?

נשים לב שההבדלים נובעים מהזמן בו מוחלף הy הראשון בפונקציה y - במודל ההחלפות ברגע שהתייחסו לy כמשתנה חופשי החלפנו אותו ב4 , ולכן הפונקציה foo עבדה עם $y = 4$. לעומת זאת במודל ה sc הכנסנו את הפונקציה foo כמו שהיא למחסנית ובשלב זה y עדיין לא נקבע. בגישה זו foo מבחינתנו הוא אובייקט בטוח, והערכים שלו נקבעים רק כאשר מתבצע *call*

ההבדל הזה למעשה מגדיר שני מודלים, דינמי וסטטי ואלו ההגדרות:

- Static Scope (also called Lexical Scope):
  - In a language with static scope, each identifier gets its value from the scope in which it was defined (not the one in which it is used).
  - הערכים מוגדרים בשלב ההגדרה.
- Dynamic Scope:
  - In a language with dynamic scope, each identifier gets its value from the scope of its use (not its definition).
  - הערכים מוגדרים בשלב השימוש.

הרצה במודל הסטטי:

```
#lang pl untyped
#lang pl dynamic
(define x 123)
(define (getx) x)
```

```
(define (bar1 x) (getx))
(define (bar2 y) (getx))
;;
(test (getx) => 123) ;;
(test (let ([x 456]) (getx)) => 123) ;; (1)
(test (getx) => 123)
(test (bar1 999) => 123)
(test (bar2 999) => 123)
```

1. הפונקציה $getX$ הוגדרה עם הערך 123 , ולכן **תמיד** תחזיר 123

אותה הרצה עם המודל הדינמי

```
#lang pl dynamic
;;
(define x 123)
(define (getx) x)
(define (bar1 x) (getx))
(define (bar2 y) (getx))
;;
(test (getx) => 123)
(test (let ([x 456]) (getx)) => 456) ;; (1)
(test (getx) => 123) ;; (2)
(test (bar1 999) => 999) ;; (3)
(test (bar2 999) => 123) ;; (4)
```

1. במודל הדינמי ה $getx$ תוגדר בעזרת הx שבlet ולכן תחזיר 456 . 2. יצאנו מה$scope$ ולכן נחזיר להגדרה הוקדמת של x . 3. ב$scope$ הזה x מוגדר עם 999 4. y מוקשר ל999 וחזרנו להגדרה המקורית של $x = 123$

המשך הדוגמה:

```
#lang pl dynamic
(define x 123)
(define (getx) x)
(define (bar1 x) (getx))
(define (bar2 y) (getx))
;;
(define (foo x) ;; (1)
    (define (helper) (+ x 1)) helper)
(test ((foo 0)) => 1)
;; and *much* worse: (2)
(define (add x y) (+ x y))
(test (let ([+ *])
```

```
(add 6 7)) => ?)
```

<u>במודל הסטטי:</u>

1. ה$helper$ בהגדרה שלה , מוגדרת על ידי $0,1$ ולכן foo תחזיר $0+1=1$
2. הגדרנו את $+$ להיות $*$, במודל הסטטי ה$+$ מוגדר כבר בזמן הגדרת הפונקציה ולכן נקבל $6+7=13$

<u>בגרסה הדינמית:</u>

1. ה$helper$ נשמרת ה $\{+ \text{ x } 1\}$ וכאשר אנחנו יוצאים מה$scope$ הx כבר לא מוגדר כ0 , ולכן כאשר קוראים ל foo היא נקראת עם $x=123$ והתוצאה היא 124
2. הגדרנו את $+$ להיות $*$, במודל הדינמי זה מה שנשתמש בקריאת הפונקציה עם ה$+$ ולכן נקבל $6*7=42$

דוגמה נוספת:

```
#lang pl dynamic
(define tax% 6.5)
(define (with-tax n)
    (+ n (* n (/ tax% 100))))
(with-tax 10) ; how much do we pay?
    (let ([tax% 17.0]) (with-tax 10)) ; how much would we pay in Israel?
;; make that into a function
(define il-tax% 17.0)
(define (us-over-il-saving n)
    (- (let ([tax% il-tax%]) (with-tax n))
    (with-tax n)))
(us-over-il-saving 10)
;; can even control that: how much would we save if the tax in israel
;; went down one percent?
(let ([il-tax% (- il-tax% 1)]) (us-over-il-saving 10))
```

בדוגמה חישוב tax באזורים שונים ־שכ"א יש לו אחוז אחר של tax , ניתן לראות שבקלות אנחנו עוברים מtax אמריקאי ל tax ישראלי. משמעות הדבר שהמודל הדינמי נותן לנו להתממשק בקלות עם תוכנה קיימת. מצד שני קל בקלות לצור באגים ־ יש לא מעט החלטות שהמתכנת מקבל תוך כדי ריצה מסיבות מסוימות ודריסה של המידע שהוא קבע יכול לצור באג. לכן רוב השפות הן סטטיות.

לכן נרצה לחזור לקוד שלנו, ולשנות אותו שיהיה עם המחסנית ושבכל זאת יהיה סטטי ־ בדוגמה מתחילת השיעור נרצה שהתוצאה תהיה 28 , איך נעשה זאת? נצטרך לטפל בשמירה של פונקציה.

0.6 $Closure$ - מודל הסביבות

נגדיר אובייקט שגם יחזיק את הפונקציה, וגם את ה sc (נראה לו בשמו המוכר env) , ובכך נפתור את הבעיה, לדוגמה עבור:

```
{with {y 4}
    {fun {x} {* x y}}
```

נגדיר:

```
closure:
    param-name : 'x
    body: (Mul (Id 'x) (id 'y)
    env: (list ('y (Num 4)))
```

לכן צריך לבצע שני שינויים:

1. לשנות את $eval$ שתחזיר את $closure$
2. לשנות את $eval$ של $call$ שתדע להתשמש ב env של ה$closure$

לכן נגדיר אובייקטים חדשים:

```
 (define-type ENV
    [EmptyEnv]
    [Extend Symbol VAL ENV])
 (define-type VAL
    [NumV Number]
    [FunV Symbol FLANG ENV])
```

בנוסף נשנה את הגדרת $lookup$:

- כעת יקבל את env ויחזיר את val
- החיפוש יהיה בריקורסיה על ENV

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

$arith-op$ צריך לבצע את ההתאמות ל Val ו Env :

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

ב $eval$ נשנה את :

- שהערך שמקבלים יהיה ENV והמוחזר יהיה Val
- ב$with$ להתאים את הקוד לקונסטרקטור ולכן $Extend$
- בFun להחזיר גם את env על ידי $FunV$ (בנאי של VAL)
- ב$Call$ נחזיר את $FunV$ עם הסביבה ה**מתאימה**, והרחבה תהיה על הסביבה ה**נוכחית** - וזה השינוי לסביבה סטטית

```
(: 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); We expect a closure
               (eval bound-body
                 (Extend bound-id (eval arg-expr env) f-env))]
             [else (error 'eval "`call' expects a function, got: ~s"
                          fval)]))]))
```

ושינוי אחרון ב run :

- run תקרא ל$eval$ עם סביבה ריקה - $EmptyEnv$
- והיא מצפה לקבל $NumV$

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

לסיכום

- הצגנו שני מודלים סטטי ודינמי
- ראינו שמודל ההחלפות הוא סטטי ו sc הוא דינמי, סיימנו במודל הסביבות הenv (סטטי)
- נריץ דוגמה על פי מודל הסביבות:

```
{with {foo {fun {x} {* x y}}}
   {with {y 6}
      {call foo 7}}}
```

  - נקבל $error$ בגלל שאין $binding$ ל y

- נריץ על פי מודל ההחלפות

```
{with {foo {fun {x} {* x y}}}
   {with {y 6}
      {call foo 7}}}
```

  - לאחר ההחלפה:

```
{with {y 6}
   {call {foo {fun {x} {* x y}} 7}}}
```

  - לאחר ההחלפה:

```
{call {foo {fun {x} {* x 6}} 7}}}
```

  - והתוצאה 42

- כלומר במודל ההחלפות היה לנו באג, שלא ידע לזהות בעיה מסוימת ותקנו אותה במודל הסביבות.

שיעור 11

0.6.1 הוספת $Pair$

בשיעור האחרון - סיימנו לכתוב את ה $interpeter$ במודל הסביבות על ידי $closure$ - אובייקט סגור.

הראנו שכאן היה השינוי המשמעותי:

```
[(Call fun-expr arg-expr)
    (let ([fval (eval fun-expr env)])
    (cases fval
        [(FunV bound-id bound-body f-env); We expect a closure
            (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
```

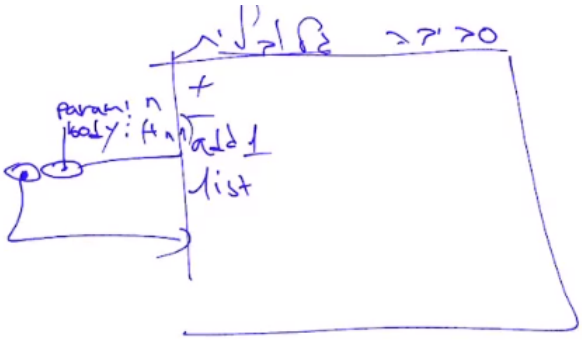
שאלה ממבחן מהו השינוי המינמלי (בקוד) כדי להעביר את הקוד מודל הסביבות למודל דינמי?

תשובה: לשנות את $f-env$ לenv , שהמשמעות היא שהenv הוא לא תלוי $scope$, כי שעושים ב $f-env$.

למבחן: צריך לדעת להסביר איך טסטים שונים ירוצו, במודל השונים.

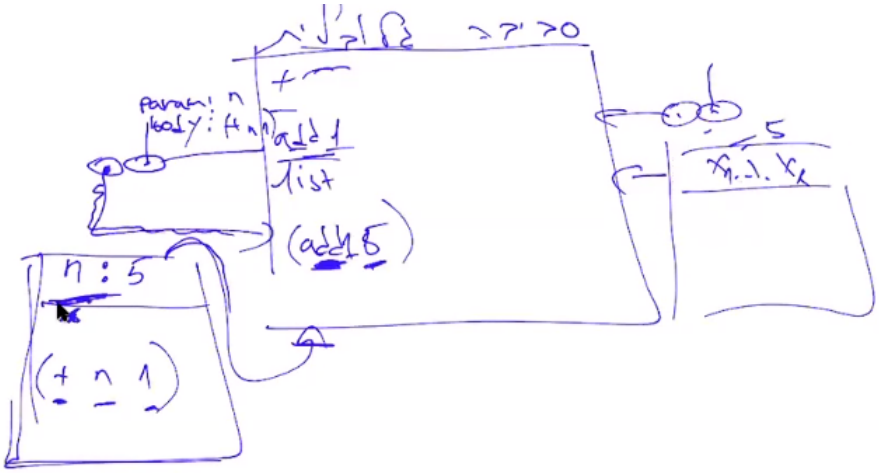
איך זה עובד ב $racket$ עצמה:

יש לנו סביבה גלובלית, שבה יש את כל הפונקציות:



קריאה לפונקציה $add1$, יוצאת לאזור שמתסכל על שני דברים:

- מי ה $param$ ו$body$
- מי הסביבה שלו - במקרה הזה הסביבה הגלובלית



- קריאה לפונקציה (add1 5) - יוצר הרחבה של הסביבה הגלובלית, הולכת ושואלת את מי אני מכיר, אצלנו יכיר את 5 ואת 1 , אבל מי זה + , את זה הוא לא מכיר, ולכן ילך ויקח את ה + , וישתמש מהסביבה הגלובלית שם יתבצע תהליך זהה של פתיחת $scope$ - ולבסוף נקבל את התצואה של $5+1$

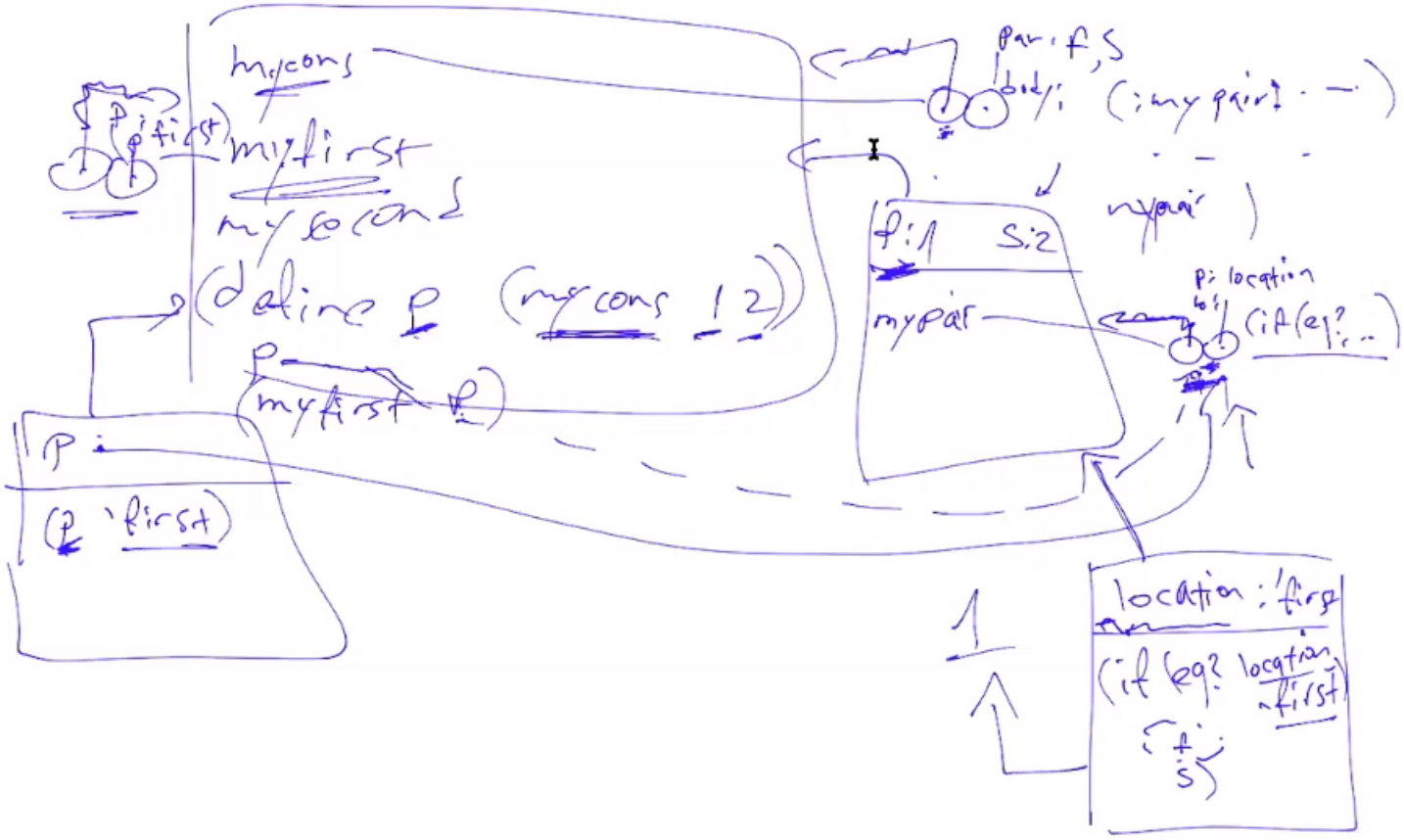
דוגמה נוספת:

נניח ונרצה לבנות מנגנון שיעבדו באופן דומה ל $cons$ שאנחנו מכירים מהרשימות, נקרא לה $mycons$ והיא תצור לנו זוגות:

- $mycons$ יקבל שני מספרים, ותחזיר פונקציה שיודעת להחזיר מספר

```
(: mycons : Number Number -> ((U 'first 'second) -> Number))
(define (mycons f s)
    (: mypair : (U 'first 'second) -> Number)
        (define (mypair location)
            (if (eq? location 'first) f s)) mypair)
;;
(: myfirst : ((U 'first 'second) -> Number) -> Number)
    (define (myfirst p)(p 'first))
;;
(: mysecond : ((U 'first 'second) -> Number) -> Number)
    (define (mysecond p)(p 'second))
;;  -- tests --
(define p (mycons 1 2))
(test (myfirst p) => 1)
(test (mysecond p) => 2)
```

מה ראינו בעצם? שעל ידי תכנות פונקציונלי אנחנו מקבלים אובייקט שאנחנו יכולים לעבוד איתו. בדיאגרמה:



במודל הדינמי זה עובד אחרת, ולמשל בקוד שלנו ה f לא תהיה מוגדרת, ותחזיר שגיאה

```
#lang pl dynamic

;(: mycons : Number Number -> ((U 'first 'second) -> Number))
(define (mycons f s)
 ; (: mypair : (U 'first 'second) -> Number)
  (define (mypair location)
    (if (eq? location 'first) f s)
  mypair)

;(: myfirst : ((U 'first 'second) -> Number) -> Number)
(define (myfirst p)
  (p 'first))

;(: mysecond : ((U 'first 'second) -> Number) -> Number)
(define (mysecond p)
  (p 'second))


(define p (mycons 1 2))
```

```
Welcome to DrRacket, version 7.6 [3m].
Language: pl, with test coverage [custom]; memory limit: 128 MB.
No tests performed!
> (test (myfirst p) => 1)

Library/Racket/7.6/collects/pl/lang/dynamic.rkt:49:9: reference
to undefined identifier: f
>
```

כעת נרצה לשפר את הקוד ־ולא להשתמש בסימבולים:

```
(: mycons : Number Number -> ((Number Number -> Number) -> Number))
(define (mycons f s)
    (: mypair : (Number Number -> Number) -> Number)
        (define (mypair loc-sel)
            (loc-sel f s)
```

```
;;
(: myfirst : ((Number Number -> Number) -> Number) -> Number) -> Number)
    (define (myfirst p)
    (: f-sel : (Number Number -> Number)
    (define (f-sel a b) a)
    (p f-sel))
;;
(: mysecond  : ((Number Number -> Number) -> Number) -> Number) -> Number)
    (define (mysecond  p)
    (: s-sel : (Number Number -> Number)
    (define (s-sel a b) b)
    (p s-sel))
;;  -- tests --
(define p (mycons 1 2))
(test (myfirst p) => 1)
(test (mysecond p) => 2)
```

ניתן גם לממש את הכל בעזרת הקוד שבנינו במהלך השיעורים על ידי החלפת ה $Number$ ב Val, ולהכניס זאת לקוד שלנו:

```
(: mycons : VAL VAL -> ((VAL VAL -> VAL) -> VAL))
(define (mycons f s)
    (: mypair : (VAL VAL -> VAL) -> VAL)
    (define (mypair loc-sel)
        (loc-sel f s))
    mypair)
(: myfirst : ((VAL VAL -> VAL) -> VAL) -> VAL)
(define (myfirst p)
    (: f-sel : VAL VAL -> VAL)
    (define (f-sel a b) a)
    (p f-sel))
(: mysecond : ((VAL VAL -> VAL) -> VAL) -> VAL)
(define (mysecond p)
    (: s-sel : VAL VAL -> VAL)
    (define (s-sel a b) b)
    (p s-sel))
```

אז נעדכן את Val :

```
(define-type VAL
    [NumV Number]
    [FunV Symbol FLANG ENV]
    [PairV ((VAL VAL -> VAL) -> VAL)])
```

נעדכן את $FLANG$

```
(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]
    [Cons FLANG FLANG]
    [First FLANG]
    [Second FLANG]
)
```

נעדכן את $parse-sexpr$:

```
(: 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))]
    [(list 'cons f s) (Cons (parse-sexpr f) (parse-sexpr s))]
    [(list 'first p) (First (parse-sexpr p))]
    [(list 'second p) (First (parse-sexpr p))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

נוסיף פונקציית $PairV \to pair$:

```
(: PairV->pair : VAL -> ((VAL VAL -> VAL) -> VAL))
(define (PairV->pair v)
    (cases v
        [(PairV p) p]
        [else (error 'PairV->pair "expects a pair, got: ~s" v)]))
```

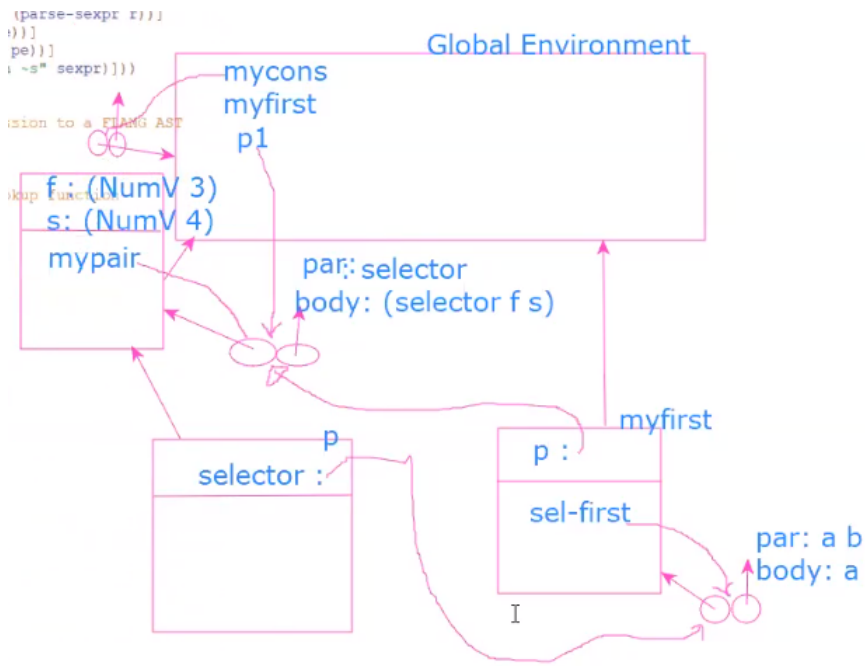
ונעדכן את $eval$:

```
(: 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); We expect a closure
               (eval bound-body
                 (Extend bound-id (eval arg-expr env) f-env))]
             [else (error 'eval "`call' expects a function, got: ~s"
                         fval)]))]
             [(Cons a b) (PairV (mycons (eval a env) (eval b env)))]
             [(First p) (myfirst (PairV->pair (eval p env)))]
             [(Second p) (mysecond (PairV->pair (eval p env)))]
        ))
```

דוגמה:

```
(test (run "{first {cons {call {with {x 3}
                       {fun {y} {+ x y}}}
                       4}
                      {fun {e} e}}}")       => 7)
```

חזרה על הסביבה הגלובלית:



0.6.2 Lazy Evaluation

שאלה: מדוע ב $eval$ בשורה המסומנת, אנחנו שולחים את $arg - exp$ ולא את arg ?

```
[(Call fun-expr arg-expr)
    (let ([fval (eval fun-expr env)])
    (cases fval
        [(FunV bound-id bound-body f-env); We expect a closure
            (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
```

נדגים זאת בטסט בהשוואה בין שתי טסטים

טסט ראשון:

```
{with { f {fun {x} {* x x}}}
    {call f {+ 4 6}}}
```

טסט שני:

```
{with { f {fun {x} {* x x}}}
    {with {z {+ 4 6}}}
```

```
{call f z}}}
```

התשובה היא:

- שאנחנו ככה מרויחים יעילות - כלומר על שליחת של ה $arg-exp$ אני בעצם שולח 10 , ולא $\{+\ 4\ 6\}$ שמחביתנו מסמל שליחת עץ - שמבחינה חישובית יכול להיות מאוד יקר.
- במודל הסביבה - אנחנו מרויחים שהביטוי מפורש לפי הסביבה, ואחרת אנו עלולים לבצע חישוב שגוי.

את הרעיון שהצגנו כעת, אפשר לפתח ולשאול - מה יקרה אם נריץ את:

```
# pl racket
;
(if 1 2 (/7 0))
```

התשובה: פשוט נקבל 2 כי 1 הוא $\#t$, ולכל לא נגיע לקוד הבעייתי שדורש חלוקה ב0

אבל אם נגדיר:

```
#lang racket
(define (myif a b c)
    (if a b c)
(myif 1 2 (/7 0))
```

אז נקבל שגיאה, כי מריצים את כל העץ = גם את c = חלוקה באפס, והיינו רוצים שגם אצלנו נוכל לעשות Lazy Evaluation , כלומר שנעריך ביטויים רק כשנדרש להם. בpl יש אופציה להגדרה שכזו ונוסיף בתחילת הקוד את:

```
#lang pl lazy
```

הדגמה נוספת:

```
(define onces (cones 1 ones))
```

ב$reackt$ סתם, זה יתן שגיאה כי אנחנו מגדירים אובייקט על האובייקט עצמו, אבל ב $lazy$ זה יתן את:

```
No tests performed!
> ones
#0='(1 . #0#)
> (first ones)
1
> (first (rest ones))
1
> (first (rest (rest (rest (rest ones)))))
1
> (second ones)
1
> (rest ones)
#0='(1 . #0#)
```

למעשה הגדרנו אובייקט אינסופי.

הדגמה נוספת:

```
(define (foo x) (+ x y)
    (let ([let y 6])
        (foo 3))
```

סיכום

ב$racket$ עצמה נקבל שגיאה - כי y לא מוגדר ב$body$ של foo

ב$dynamic$, לא תהיה בעיה כי היא תדע לחבר את הyל 6

ב $pl\ lazy$, לא יצליח לבצע, כי זה שאמרנו לו להיות $lazy$ לא אומר שעברנו לשיטה של ל $dynamic$, כלומר עדיין צריך ש

- בהגדרת הפונקציה ה y יהיה מגודר.
- כל אובייקט צריך להיות מחושב פעם אחת

~~ חזרה על הנושאים שהיה בקורס ~~

מספר הארות:

- $racket$ - שפה דינמית
- pl - שפה סטטית
- השפה שבנינו $FLANG$ - היא דינמית (לא מבקשת הצהרה על משתנה)
- מודל הסביבות - סטטי

נספח

1 הגדרות

כללי

1. רקורסית זנב: חישוב ושליחה חוזרת עד לתנאי עצירה בדרך כלל רקורסיית זנב דורשת פונקציית עזר
2. ambiguty - קיימת מילה בשפה שניתן לגזור אותה לפי שני עצי גזירה שונים
3. syntactic sugar: מקרה בו יש לי בשפה פתרון לפעולה מסוימת אבל אני רוצה לפשט אותה (="להמתיק") ולכן ממציא תחביר חלופי שעושה את אותו דבר.

1.0.1 $with$

הגדרה פורמלית Compositionality:

$eval$ על קודקוד T עם בנים $T_1, ..T_k$ יבצע :

1. $v_1 = eval\left(T_1\right), v_2 = eval\left(T_2\right) ....v_k = eval\left(T_k\right)$
2. יבצע פעולה מקומית על $v_1, ..., v_k$

החלפת משתנים הגדרות

- Binding instance - מופע של x , שבו אני מקשר את x לביטוי כלשהו (הצהרה על שם מזהה x)
- Scope - עבור Binding instance אותו חלק בקוד שבו כל מופע של x מקושר למופע ההצהרתי הזה
- Bound instance - כל מופע של x שאינו Binding instance וגם נמצא בתוך $scope$ של Binding instance
- Free instance - כל מופע של x שאינו Binding instance וגם אינו Bound

$$\{+ \underbrace{x}_{\substack{\text{free}\\ \text{incstance}}} \quad \{\text{with } \{ \underbrace{\text{x}}_{\substack{\text{binding}\\ \text{incstance}}} \underbrace{5}_{\substack{\text{value}\\ \text{bound}}} \}\overbrace{\{ + \underbrace{\text{x}}_{\substack{\text{bound}\\ \text{instance}}} 1\}}^{\text{scope}}\}\}$$

סימונים

החלף את המופעים המתאימים של $name\left(=x\right)$ בערך v בתוך $body\left(=e\right)$

1. הסינטקס: [name/v]body
2. : פורמלית: $[x/v]\, e$

1.0.2 פונקציות

1. פונקציית (first order) - אובייקט שיודע לבצע חישוב, לא יכול לשמור $data$
2. פונקציית (high order) - אובייקט יותר מורכב שניתן להביע איתו חישובים ותהלכים ברמה גבוה וקריאה, בעל אפשרות של שליחה וקבלת פונקציה כמשתנה.

3. פונקצית (First class) - שפות בהן פונקציה היא טיפוס כמו כל טיפוס אחר ⇐ נותן לנו את האפשרות לפונקציה ללא שם (אנונימית)

(א) מאפשר להגדיר פונקציה בזמן ריצה

(ב) יכולות להשמר במבנה נתונים, ועוד.

- Static Scope (also called Lexical Scope):
  - In a language with static scope, each identifier gets its value from the scope in which it was defined (not the one in which it is used).

– הערכים מוגדרים בשלב ההגדרה.

- Dynamic Scope:
  - In a language with dynamic scope, each identifier gets its value from the scope of its use (not its definition).

– הערכים מוגדרים בשלב השימוש.

```
closure:
    param-name : 'x
    body: (Mul (Id 'x) (id 'y)
    env: (list ('y (Num 4)))
```

2 הערות $parse$

1. $With$

(א) $name$

(ב) $val\backslash named$

(ג) $body$

2. Fun :

(א) $name\backslash param$

(ב) $body$

3. $Call$:

(א) $fun\ name$

(ב) $fun\ arg$

3 אלגוריתמים ב $eval$

r : היא הקריאה ה r

הערה חשובה: המספר $r, r+1, r+2$ הוא של פעולות נדרשות ביחס לבנאי מסוים, אבל בעצים מורכבים יתכנו הפרשים גדולים יותר במחסנית הקריאות

מודל ההחלפה:

1. $AST_r = (Num\ n)$:
   - נחזיר את $(Num\ n) \iff Res_r = (Num\ n)$
2. $AST_r = (Id\ name)$
   - בשלב זה אסור לקבל $Id \iff$ נחזיר שגיאה
3. $AST_r = (OP\ l\ r)$
   - גורם לשתי קריאות נוספות ל $eval$
     - נקרא ל $eval\,(l) \iff AST_{r+1}\,(l)$
     - נקרא ל $eval\,(r) \iff AST_{r+2}\,(r)$
   - Res_r מחכה לתוצאה
4. $AST_r = (Fun\ b-id\ b-body))$
   - מחזיר את הפונקציה $\iff RES_r = (Fun....)$
5. $AST_r = (With\ b-id\ named\ b-body)$
   - גורם לשתי קריאות נוספות ל $eval$
   - גורם לקריאת $Subst$ על ה$b-body$
     - נקרא ל $eval\,(named) \iff AST_{r+1}\,(named)$
     - נקרא ל $subst$ עם $\left\{\begin{matrix} b-body \to expr \\ b-id \to from \\ eval\,(named) \to to \end{matrix}\right\}$, החלפה תתבצע ל:
       * $named$
       * $b-body$ רק אם $b-id \neq from$ $\Leftrightarrow$ כן להחליף + לא לדרוס $Scope$ עם שם משתנה דומה $\Leftrightarrow$ רק את ה $bound$ שנמצא בתוך ה $scope$)
     - קרא ל $eval\,(b-body)$ על ה $body$ <u>המעודכן</u> $\iff AST_{r+2}\,(b-body)$
   - Res מחכה לתוצאה
6. $AST_r = (Call\ f-expr\ arg))$
   - גורם לשלוש קריאות נוספות ל $eval$
   - קריאה ל$subst$ על ה $b-body$
     - וידוא ש $f-expr$ הוא פונקציה $eval\,(f-expr\ env_r) \iff AST_{r+1}\,(f-expr\ env_r)$
       * אם לא - מחזירים שגיאה
       * אם כן נשתמש בפרמטים $b-id, b-body$, ו:
     - נקרא ל $eval\,(arg)$ עם $env_r \iff AST_{r+2}\,(arg\ env_r)$
     - נקרא ל $subst$ עם $\left\{\begin{matrix} b-body \to expr \\ b-id \to from \\ eval\,(arg) \to to \end{matrix}\right\}$, החלפה תתבצע ל:
       * arg
       * $b-body$ רק אם $b-id \neq from$ $\Leftrightarrow$ כן להחליף + לא לדרוס $Scope$ עם שם משתנה דומה $\Leftrightarrow$ רק את ה $bound$ שנמצא בתוך ה $scope$)
     - נקרא ל $eval\,(b-body)$ על ה $body$ <u>המעודכן</u> $\iff AST_{r+3}\,(b-body)$

מודל הSC:

1. $AST_r = (Num\ n)$:
   - נחזיר את n $\iff$ $Res_r = n$
   - SC לא משתנה
2. $AST_r = (Id\ name)$
   - נחזיר את $lookup\,(name)$ עבור ה SC_r המתאים (מושפע מעץ הקריאות) $\iff$ $Res_r = lookup\,(name\ SC_r)$
   - SC לא משתנה
3. $AST_r = (OP\ l\ r)$
   - גורם לשתי קריאות נוספות ל $eval$
     - נקרא ל $eval\,(l)$ עם SC_r $\iff$ $AST_{r+1}\,(l)$
     - נקרא ל $eval\,(r)$ עם SC_r $\iff$ $AST_{r+2}\,(r)$
   - Res_r מחכה לתוצאה
   - SC לא משתנה
4. $AST_r = (Fun\ b-id\ b-body))$
   - מחזיר את הפונקציה $\iff$ $RES_r = (Fun....b-body)$
   - SC_r לא משתנה
5. $AST_r = (With\ b-id\ named\, b-body)$, עם SC_r
   - גורם לשתי קריאות נוספות ל $eval$
     - נקרא ל $eval\,(named\ env_r)$ $\iff$ $AST_{r+1}\,(named)$
     - נרחיב את את SC_r במעבר ל AST_{r+2} עם הזוג: ($b-id$, התוצאה על $named$)
     - נקרא ל $eval\,(b-body\ SC_{r+2})$ $\iff$ $AST_{r+2}\,(b-body)$
   - Res_r מחכה לתוצאה
6. $AST_r = (Call\ f-expr\ arg))$ עם SC_r
   - גורם לשלוש קריאות נוספות ל $eval$
     - וידוא ש $f-expr$ הוא פונקציה $eval\,(f-expr\ sc_r)$ $\iff$ $AST_{r+1}\,(f-expr\ sc_r)$
       - אם לא - מחזירים שגיאה
       - אם כן נשתמש בפרמטים $b-id, b-body$, ו:
     - נקרא ל $eval\,(arg)$ עם sc_r $\iff$ $AST_{r+2}\,(arg\ sc_r)$
     - נרחיב את sc_r במעבר ל ENV_{r+3} עם הזוג: ($b-id$, התוצאה על arg)
     - נקרא ל $eval\,(b-body\ Extend\,(SC))$ $\iff$ $AST_{r+3}\,(b-body)$
   - Res מחכה לתוצאה

מודל הסביבות:

1. $AST_r = (Num\ n)$:
   - נחזיר את $(NumV\ n)$, כלומר $Res_r = (NumV\ n)$
   - ENV לא משתנה
2. $AST_r = (Id\ name)$
   - נחזיר את $lookup\,(name)$ עבור ה ENV_r המתאים (מושפע מעץ הקריאות) $\iff Res_r = lookup\,(name\ env_r)$
   - ENV לא משתנה
3. $AST_r = (OP\ l\ r)$
   - גורם לשתי קריאות נוספות ל $eval$
     - נקרא ל $eval\,(l)$ עם ENV_r $\iff AST_{r+1}\,(l)$
     - נקרא ל $eval\,(r)$ עם ENV_r $\iff AST_{r+2}\,(r)$
   - Res_r מחכה לתוצאה
   - ENV לא משתנה
4. $AST_r = (Fun\ b-id\ b-body))$
   - מחזיר את הפונקציה <u>עם הenv_r</u> $\iff RES_r = (FunV....env_r)$
   - ENV_r לא משתנה
5. $AST_r = (With\ b-id\ named\,b-body)$, עם env_r
   - גורם לשתי קריאות נוספות ל $eval$
     - נקרא ל $eval\,(named\ env_r)$ $\iff AST_{r+1}\,(named)$
     - נרחיב את את ENV במעבר ל ENV_{r+2} עם התוצאה על $named$
     - נקרא ל $eval\,(b-body\ env_{r+1})$ $\iff AST_{r+2}\,(b-body)$
   - Res מחכה לתוצאה
6. $AST_r = (Call\ f-expr\ arg))$
   - גורם לשלוש קריאות נוספות ל $eval$
     - וידוא ש $f-expr$ הוא פונקציה $eval\,(f-expr\ env_r)$ $\iff AST_{r+1}\,(f-expr\ env_r)$
       - אם לא - מחזירים שגיאה
       - אם כן נשתמש בפרמטים $b-id, b-body.f-env$, ו:
     - נקרא ל $eval\,(arg)$ עם env_r $\iff AST_{r+2}\,(arg\ env_r)$
     - <u>נרחיב את $f-env$</u> במעבר ל ENV_{r+3} עם התוצאה על arg
     - נקרא ל $eval\,(b-body\ Extend\,(f-env))$ $\iff AST_{r+3}\,(b-body)$
   - Res מחכה לתוצאה