אוניברסיטת
אריאל
בשומרון

תקציר לקורס שפות תכנות

נכתב ע"י צבי מינץ zvimints@gmail.com

מבוסס על הרצאות של ד"ר עמרי ערן **ועל הסיכום שמופיע במודל של שנים קודמות**

שיעור 1:

מהן יסודות של השפות?

1. **תחביר (Syntax):**
   מערכת הכללים של השפה
2. **סמנטיקה (Semantics):**
   משמעיות, כגון: בהינתן תחביר אשר שייך לשפה אזי לדעת מה המשמעות העומדת מאחורי המילה בשפה

ההבדל במשמעות בין תחביר לסמנטיקה:

התחביר חשוב בעיקר לצורכי "קוסמטיקה". הוא נועד לשרת את הסמנטיקה, הסמנטיקה **חשובה יותר** מהתחביר, שכן את התחביר אפשר לשנות.

לדוגמא:

- אם נסתכל על המספר 42, שכתוב בקובץ כלשהו – מדובר בעצם בייצוג בין שני ערכים ASCII, לעומת המס' 42 שמאוחסן בזיכרון.
- המילה "שוד" היא סתם מילה, אבל אם נחשוב על המשמעות שלה – אפשר לשבת בכלא בגלל ביצוע השוד

דוגמא לחשיבות של הסמנטיקה:

מערכך כלשהו שמאותחל ל-X תאים, כאשר נבקש בתוכנית לגשת לתא ה-$X+1$ אזי בכל שפה התחביר הוא שונה, אבל נקבל את אותה התוצאה שהיא – לא ניתן גלשת לתא הזה כי הוא אינו קיים בעצם.

מכאן המסקנה כי התחביר נועד לצורכי "קוסמטיקה" בלבד וניתן לשנות אותו, **וסמנטיקה** הרבה יותר חשוב מהתחביר כי היא בעצם **המשמעות**

הערה: על מנת להסביר היטב מהי סמנטיקה, נשתמש בתוכנית.

נשתמש ב-Racket מסיבות מגוונות כמו:

תחביר, פונקציונאליות, סביבת עבודה וכו', פונקציית הערה שבה ראקט משתמשת היא למעשה פונקציה שמקבלת סינטקס ומחזירה או מבצעת את הסמנטיקה שלו.

הערה: קומפיילר לוקח תוכנית בשפה עלית ומתרגם אותה לשפה תחתית יותר, ואז בעצם אם ידועים את הסמנטיקה של שפת C אז הוא נותן סמנטיקה לשפת ++C, כי משתמשים ב-C כדי לתרגם את ++C.

בקורס הזה אנחנו נבנה **Interpreter**, ההבדל הוא שקומפיילר הוא מתרגם משפה גבוהה ליותר נמוכה עד שמגיעים לשפת מכונה ואז בשביל להריץ את הקוד נטען אותו לזיכרון ונריץ אותו בשפת מכונה.

לעומת זאת **Interpreter** תוך כדי שהוא מתרגם את הקוד אז הוא גם מעביר אותו להרצה ממש.

מבוא ל-Racket:

- שפת ה-Racket שייכת למשפחת ה- Lisp (שפות פונקציונאליות)
- ליבת השפה ממומשת בשפת C.
- הטיפוסים בשפה הינם **דינמיים**, כלומר לא ניתן לדעת מראש איזה טיפוס הפונקציה מקבת ומחזירה
- השפה Racket היא **שפת תכנות פונקציונאלית** ולכן לכל חישוב שנעשה יש ערך חזרה, ולכן אם נסתכל על הקטע הקוד הבא נקבל שגיאה:

```
> (if #t "fix")
if: missing an "else" expression in: (if #t "fix")
```

היות ואין ערך **אחר (Else)**

בשפה אין ערך חזרה Void

S-Expressions

דרך לייצג רשימה מקוננת של נתונים, הוא מוגדר באופן הבא:

1. ביטוי טרמינל (סופי)
2. ביטוי מהצורה $X\ Y$ כאשר X הוא S-Expression וגם Y הוא S-Expression

כאשר החלק הרקורסיבי השני הוא ההגדרה של זוג סדור – שזה אומר שזה עצים בינאריים

דוגמאות הרצה:

```
(test (string->sexpr "zvi") => 'zvi)
(test (string->sexpr "{ zvi }") => '(zvi))
```

```
(test (string->sexpr "zvi-mints") => 'zvi-mints)
(test (string->sexpr "{+ 3 4}") => '(+ 3 4))
(test (string->sexpr "{{+ 3 4} + 5 }") => '((+ 3 4) + 5))
(test (string->sexpr "{ last semester finally {+ 1 0} + 0 }") => '(last semester finally (+ 1 0) + 0))
```

שיעור 2:

```
#lang pl
;; not
(test (not true) => false)

;; ari
(test (+ 1 3) => 4)

;; conditions (only #f is false)
(test (if true 1 -1) => 1)
(test (if #t 1 -1) => 1)
(test (if "false" 1 -1) => 1)
(test (if 0 1 -1) => 1)
(test (if null 1 -1) => 1)
(test (if 1 1 -1) => 1)

;; cond
(define n 91)
(test
 (cond
  [(<= n 9) 1] ;; test-1
  [(<= n 90) 2] ;; test-2
  [else ">=90"] ;; else-exp
) => ">=90")
(test (cond [(null? n) "1"] [(eq? n 90) "2"] [(eq? n 91) "3"]) =>
(number->string 3))

;; define (binding)
(define PI 3.14)

;; define-function
(: f : Number -> Number)
(define (f arg) (+ arg 1))

;; define-type
(define-type TREE
  [Leaf (U Number Null)]
  [Node Number TREE TREE]
)

;; list -> or empty list '() / null or a pair (pair) s.t the second
element is list
(test null => '())
(test (list 1 2 3) => '(1 2 3)) ;; (1 (2 (3 null)))
(test (cons 1 '(2)) => '( 1 2))
(test (cons 1 null) => '(1))
(test (cons 1 (list 2 3 4)) => '(1 2 3 4))
(test (list 'x 'y 'z) => (cons 'x (cons 'y (cons 'z null))))
(test (list 'x 'y 'z null) => (cons 'x (cons 'y (cons 'z (cons null
null)))))

(test (append (list 1 2) null) => (list 1 2))
(test (append (list 1 2) null) => '(1 2))
(test (append (list 1 2 null) null) => '(1 2 ()))
(test (append (list 1 2 'null) null) => '(1 2 null))
```

נכתב ע"י צבי מינץ zvimints@gmail.com


```
(test (first (list 1 2 3)) => 1)
(test (rest (list 1 2 3)) => '(2 3))

;; cases
(test
 (cases (Node 3 (Leaf 0) (Leaf null))
   [(Node current left right) right]
   [else "Damn"]
  ) => (Leaf '()))

;; template - generic type
(: every? : (All (Type) (Type -> Boolean) (Listof Type) -> Boolean))
(define (every? f list)
  (or (null? list)
      (and (f (first list)) (every? f (rest list)))
      ))
(: isPos : Number -> Boolean)
(define (isPos x) (if (>= x 0) true false))
(test (every? isPos '( 2 4 6)) => #t)
(test (every? isPos '( 2 4 -6)) => #f)

;; lambda (lambda (arg*) (body))
(test (map
       (lambda (arg)
         (add1 arg)
       )
       '(1 2 3)) => '(2 3 4))
;; let (let ([x 5]) x)
(test (let ([tree (Node 3 (Leaf null) (Leaf null))]) (Node -1 tree tree))
=> (Node -1 (Node 3 (Leaf '()) (Leaf '())) (Node 3 (Leaf '()) (Leaf
'()))))
```

הערות:

- כל דבר חוץ מערך False נחשב לערך אמת ב-Racket
- בשפת Racket אין תופעות לוואי (Side Effects)

הערות לגבי רקורסיה:

רקורסיה רגילה:

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

(test (length '(1 2 3)) => 3)
```

הבעיה בגישה זו שבעצם אפשר להגיע למצב של Stack Overflow (**הצפת זיכרון**) מכוון שהמחסנית עלולה להתמלא בקריאות חזרה.

על מנת לפתור את זה יש את **רקורסיבית זנב**, שהיא לא תצטרך לחזור בסוף התהליך אלא ישר להחזיר ערך ברגע שנגיע לתנאי העצירה

```
(: length : (Listof Any) -> Natural)
(define (length lst)
  (: helper : Natural (Listof Any) -> Natural)
  (define (helper num lst)
    (if (null? lst) num (helper (add1 num) (rest lst))))

 (helper 0 lst)
)

(test (length '(1 2 3)) => 3)
```

אוניברסיטת אריאל בשומרון

תקציר לקורס שפות תכנות

נכתב ע"י צבי מינץ zvimints@gmail.com

מבוסס על הרצאות של ד"ר עמרי ערן **ועל הסיכום שמופיע במודל של שנים קודמות**

שיעור 3:

מודל:

```
#|
S3 - parser:
<AE> ::= <num>
        { + <AE> <AE> }
        { - <AE> <AE> }
|#

(define-type AE
        [Num Number]
        [Add AE AE]
        [Sub AE AE])
```

תרשים זרימה של הפונקציות בתוכנית:

מטרה: יצירת עץ סינטקס אבסטרקטי AST

פונקציה: parse

חותמת: String → AE

קלט: מחרוזת לדוגמא "{ 5 { 4 3 + } - }"

↓ קוראת ל

פונקציה: parse-sexpr

חותמת: Sexpr → AE

↓

פלט:

(Sub (Add (Num 3) (Num 4)) (Num 5))

מטרה: חישוב ערך

פונקציה: run

חותמת: String → Number

קלט: מחרוזת לדוגמא "{ 5 { 4 3 + } - }"

↓ קוראת ל

פונקציה: eval

חותמת: Sexpr → Number

קלט: מחרוזת לדוגמא "{ 5 { 4 3 + } - }"

↓

פלט:

2

קטע הקוד:

```
#lang pl
#|
S3 - parser:
<AE> ::= <num>
        { + <AE> <AE> }
        { - <AE> <AE> }
|#

(define-type AE
        [Num Number]
        [Add AE AE]
        [Sub AE AE])

(: parse-sexpr : Sexpr -> AE)
  (define (parse-sexpr expr)
        (match expr
        [(number: num) (Num num)]
        [(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]
        [(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]
```

```
        [_ (error 'parse-sexpr "bad syntax in ~s" expr)]))


(: parse : String -> AE)
  (define (parse code)
        (parse-sexpr(string->sexpr code))
)

#|
This function ios responsible to convert sexpr to number
|#
(: eval : Sexpr -> Number)
    (define (eval expr)
        (match expr
        [(number: num) num]
        [(list '+ l r) (+ (eval l) (eval r))]
        [(list '- l r) (- (eval l) (eval r))]
        [_ (error 'eval "bad syntax in ~s" expr)]))

(: run : String -> Number)
   (define (run code)
        (eval (string->sexpr code)))

;;; Tests:
(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}}") =error> "bad syntax")


(test (run "5") => 5)
(test (run "{ + 3 4 }") => 7)
(test (run "{ - { + 3 4 } 5 }") => 2)
(test (run "{ + 4 5 { - 4 5 4}}") =error> "bad syntax")
```

הבעיה: בעיה תיכנותית ולא מודלרית
הפעולה של ניתוח סינטקטי שזה שה-Parsing, שזה להבין את המבנה של הקוד ולבדוק שהקוד בנוי נכון.
הפעולה השנייה היא איוולואציה, שזה חלק נפרד – מקבל בתור קלט את התוצר של החלק הראשון ואז מבצע עליו איוולואציה, שזה התוצר של החלק הסינטקטי (כל המשמעות של עץ סינטקס אבסטרקטי)
ואז מה שקורה שאם מחר משנים אם הסינטקס, הפונקצייה של האווקואציה לא צריכה להשתנות בעצם.

ולכן, יש פה בעיה מתוכננת בקוד מבחינה תכנותית כי **eval** מקבלת Sexpr במקום AE – **המשך בשיעור רביעי**, הסיבה שזה בעיה זה בגלל שאם

שיעור 4:
על מנת לפתור את הבעיה שמוצגת בהרצאה 3 נעשה את השינויים הבאים:

```
(: run : String -> Number)
  (define (run code)
     (eval (parse code))
)
```

תקציר לקורס שפות תכנות
נכתב ע"י צבי מינץ zvimints@gmail.com
מבוסס על הרצאות של ד"ר עמרי ערן **ועל הסיכום שמופיע במודל של שנים קודמות**

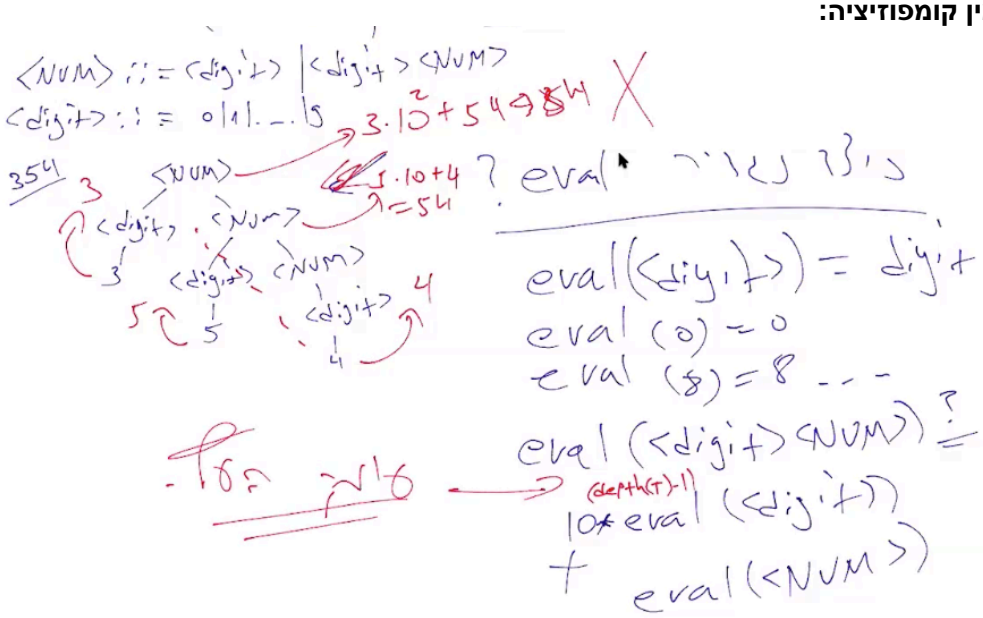
```
(: eval : AE -> Number)
  (define (eval expr)
    (cases expr
      [(Num n) n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]))
```

תכונות:

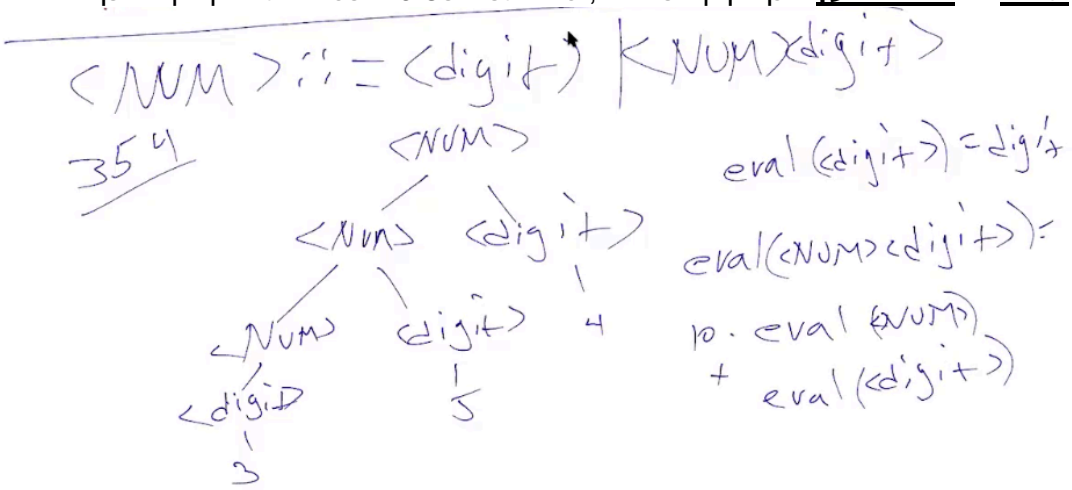
קומפוזיציה (Compositionality):

תכונה עבור דקדוק, אם קיימת פונקציית Eval כך שלכל קודקוד פנימי T עם בנים $T_1, \dots, T_n$ נחשב $eval(T) = f\left(eval(T_1), \dots, eval(T_n)\right)$ ובפרט לא צריך לדעת את מבנה הביטוי של T ו-f היא פונקצייה קבועה

דוגמא מתי אין קומפוזיציה:

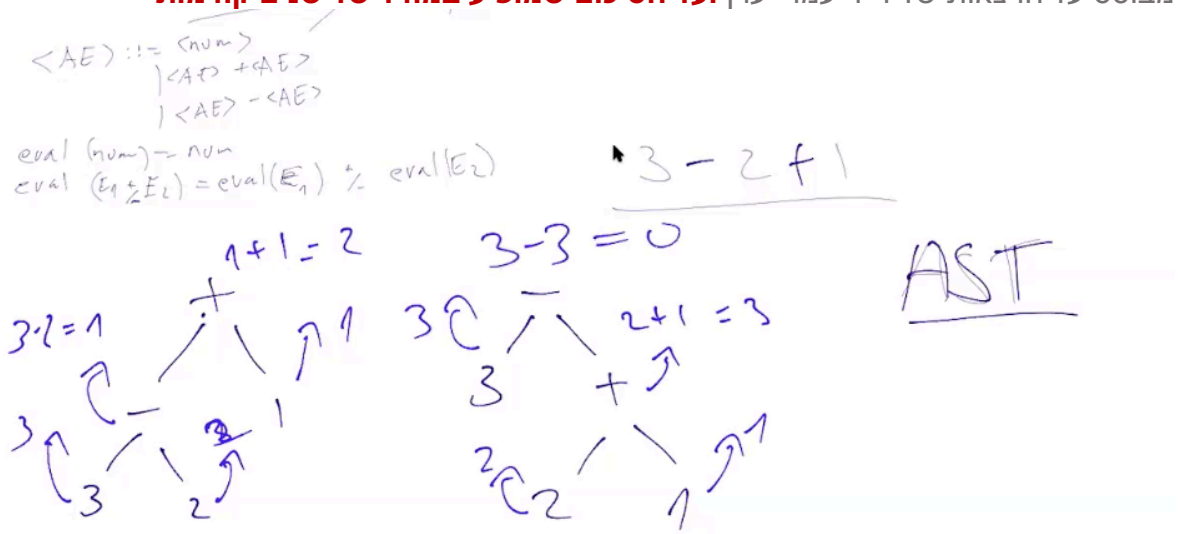


כי Eval **תלויה** גם **במבנה העץ** ולכן אין קומפוזיציה, כדי לתפור פשוט הופכים את הדקדוק באופן הבא:



חשיבות החד משמעיות (Ambiguity):

כשאין:



שפת תכנות **לא יכולה** לעבוד כך, שאחד כותב ויוצא X והשני כותב Y ו-$X \neq Y$
פתרון הבעיה אצלנו: הבעיה נפתרה ברגע שהוספנו סוגריים מסולסלים אז לא יהיה יותר רב משמעיות

שיעור 5:
מטרה: רוצים למנוע חישובים חוזרים של אותו ערך מספר פעמיים, דבר זה פוגע ביעילות של הקוד שלנו לדוגמא בביטוי:

$$\{+\ \{*\ 7\ 3\}\ \{*\ 7\ 3\}\}$$

יתרונות לניתנת שם לקטע קוד:

- קוד קצר יותר
- קוד קריא יותר
- עץ גזירה קצר יותר
- קוד יעיל יותר – הימנעות מחישובים כפולים
- נתינת שמות משמעותיים – מה שעוזר למתכנת להבין למה הוא מתייחס
- מניעת באגים הנובעים מהעתקות חוזרות של קטע הקוד

מטרה סינטקטית: נרצה שקטע הקוד הבא יעבוד:

```
{with { val { * 7 13 }
        { + val val }}}
```

אדום name
ירוק named-expression
כחול body

הערכה הכללית:

1. פעולת כפל $\{*\ 7\ 3\} \rightarrow 21$
   כלומר הערך את ה-named-expression וקבל בחזרה את הערך
   הערה: באופן כללי ניתן להחליף את ה-name ב-named-expression ללא הערכה, מה שזה אופן שכל פעם שמופיע name בתוך ה-body נצטרך להחליף את ה-name ב--named expression, הבעיה פה ש-named-expression הוא תת עץ ואם ה-named מופיע יותר מפעם אחת אז אם נלך לפי שיטה זו, ייווצר מצב של חישובים עודפים כי כל פעם נחשב את תת העץ הזה אשר הפלט שלו זהה ולכן זה לא יעיל, ולכן נחשב קודם ואז נבצע החלפה
2. החלפה (substitution) $\{+val\ val\}[val/21]$
3. חיבור $\{+21\ 21\} \rightarrow 42$

שינויים בקוד:

```
(define-type WAE
    [Num Number]
    [Id Symbol]
    [Add WAE WAE]
```

```
      [Sub WAE WAE]
      [Mul WAE WAE]
      [Div WAE WAE]
      [With Symbol WAE WAE])
```

```
(: parse-sexpr : Sexpr -> WAE)
  (define (parse-sexpr expr)
      (match expr
      [(number: num) (Num num)]
      [(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) (Div (parse-sexpr l) (parse-sexpr r))]
      [(list '* l r) (Mul (parse-sexpr l) (parse-sexpr r))]
      [(cons 'with _)
       (match expr
         [(list 'with (list (symbol: name) named-expr) body)
          (With name (parse-sexpr named-expr) (parse-sexpr body))]
         [_ (error 'parse-sexpr "bad with syntax in ~s" expr)])]
```

קטע קוד סופי:

```
#lang pl
#|
S3 - parser:
<WAE> ::= <num>
      { + <WAE> <WAE> }
      { - <WAE> <WAE> }
      { / <WAE> <WAE> }
      { * <WAE> <WAE> }
      { with { <id> <WAE> } <WAE>}

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

|#

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

(: parse-sexpr : Sexpr -> WAE)
  (define (parse-sexpr expr)
      (match expr
      [(number: num) (Num num)]
      [(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) (Div (parse-sexpr l) (parse-sexpr r))]
      [(list '* l r) (Mul (parse-sexpr l) (parse-sexpr r))]
      [(cons 'with _)
       (match expr
         [(list 'with (list (symbol: name) named-expr) body)
          (With name (parse-sexpr named-expr) (parse-sexpr body))]
         [_ (error 'parse-sexpr "bad with syntax in ~s" expr)])]
      [_ (error 'parse-sexpr "bad syntax in ~s" expr)]))


(: parse : String -> WAE)
  (define (parse code)
      (parse-sexpr(string->sexpr code))
)

;;; Tests
(test (parse "{ with {x {* 4 2}} {* x x}}") => (With 'x (Mul (Num 4) (Num 2)) (Mul (Id 'x) (Id 'x))))
```

הערה: נשאר כעת להגדיר את Eval אשר מחשבת את הביטוי עם ה-With החדש

<u>הגדרות פורמליות:</u>

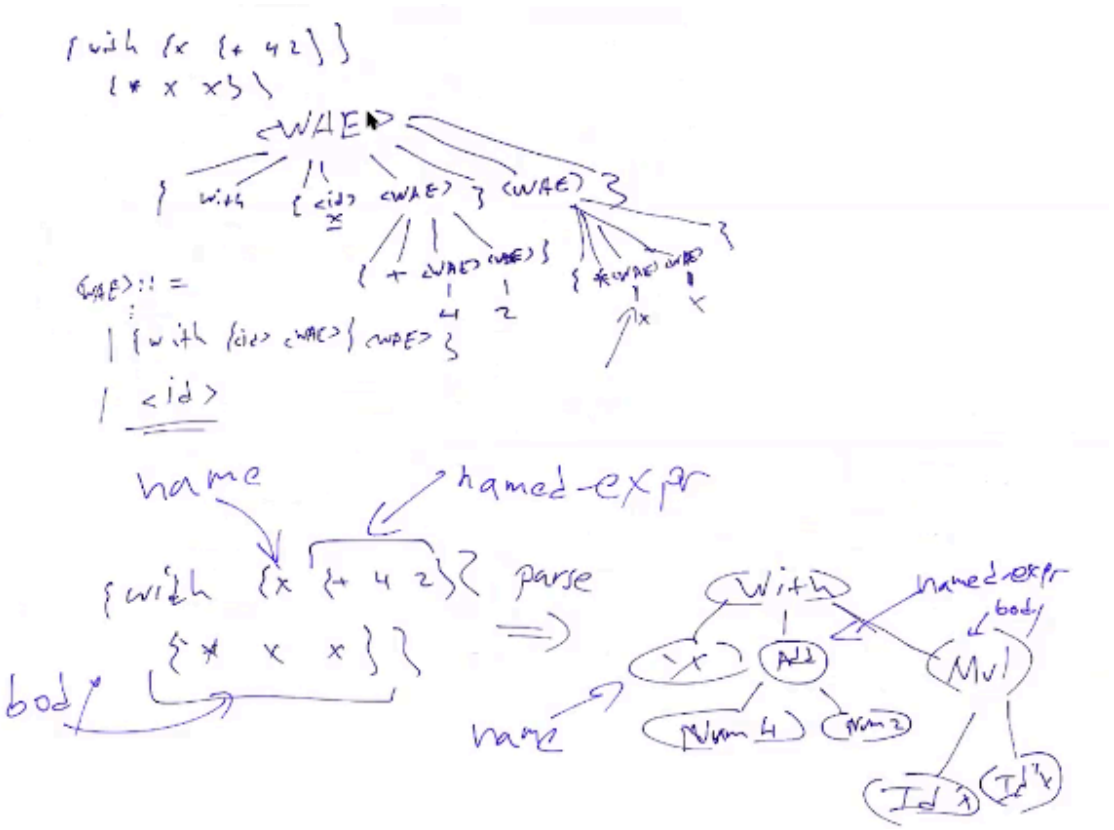
BNF – כללי הדקדוק

AST – עץ סינטקס אבסטרקטי, עץ אשר מגדיר את המבנה הסנטקטי של קטע קוד כלשהו שנכתב בשפת תכנות

<u>נתמקד כעת על שלב 2 – ההחלפות:</u>

מטרה: רוצים להגדיר פורמלית את $[x/v]e$

תמונת מצב התחלתית:



אוניברסיטת
אריאל
בשומרון

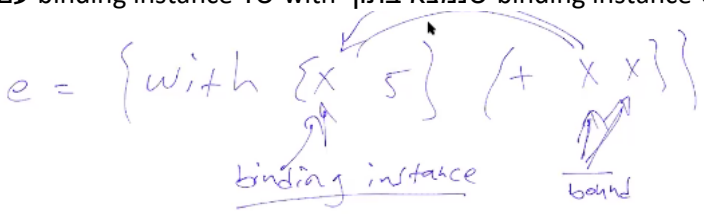
תקציר לקורס שפות תכנות

נכתב ע"י צבי מינץ zvimints@gmail.com

מבוסס על הרצאות של ד"ר עמרי ערן **ועל הסיכום שמופיע במודל של שנים קודמות**

מושגים:

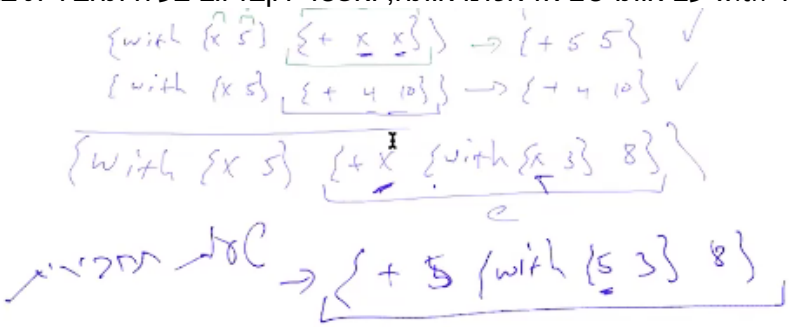
- **החלפה** $[x/v]e$: מחליפים את כל המופיעים המתאימים של x בערך v בתוך הביטוי e
- מופע של מזהה נקרא **binding instance**, משמש להצרה על שם חדש (הצהרה של שם משתמש)
  $\{with\{x\ 5\}\{+\ x\{with\{x\ 3\}\ 8\}\}$
- **scope** – האזור בקוד ביחס ל-binding instance, שהמופעים של אותו שם **קשורים** אליו
- **bound instance** – מופיע v שאינו binding instance ונמצא ב-scope של המופע binding של v, או בשפת העם זה פשוט binding instance שנמצא בתוך with של binding instance עם אותו שם



- **free instance** – מופע שאין binding ואינו bound (משתמשים חופשיים)

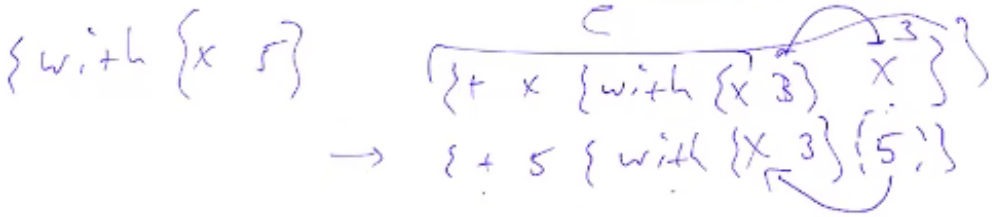
ניסיון ראשון: החלף את כל המופעים של x בתוך e בערך v

הבעיה: שאם יש עוד with עם אותו שם אז אכלנו אותה, ואפשר לקבל גם בעיה תחבירית בגלל זה



ניסיון שני: החלף את כל המופעים של x בתוך e שאינם binding instance בערך v

בעיה: עכשיו לא יהיה בעיה תחבירית אבל עדיין אם יש with עם אותו שם והערך v שונה אז פחות טוב



ניסיון שלישי: החלף את כל המופעים של x בתוך e שאינם binding instance, **ואינם בתוך ה-scope של מזהה** אחר בערך v

בעיה: אין בעיה, זה עובד, אבל נעבור לניסיון רביעי בעל הגדרה מקוצרת יותר

ניסיון רביעי: החלף את כל המופעים של x בתוך e שהם free instance בערך v

כעת נממש את פונקציית subst שמבצעת את הניסון הרביעי

אוניברסיטת
אריאל
בשומרון

תקציר לקורס שפות תכנות

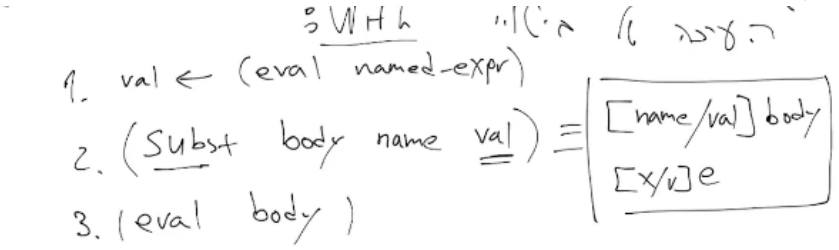
נכתב ע"י צבי מינץ zvimints@gmail.com

מבוסס על הרצאות של ד"ר עמרי ערן **ועל הסיכום שמופיע במודל של שנים קודמות**

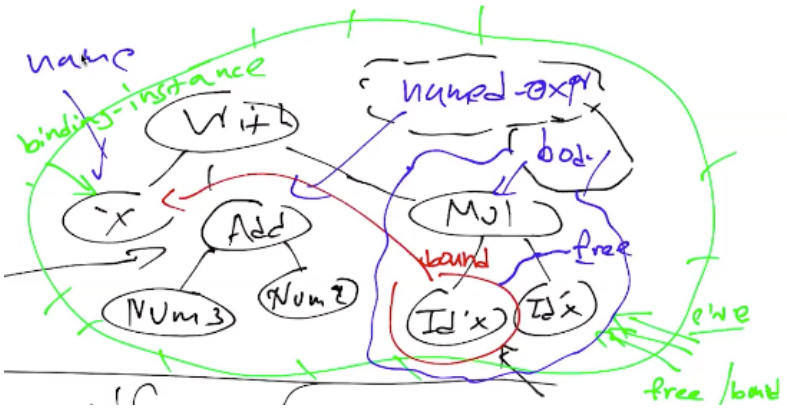
שיעור 6:

מטרה: לכתוב את subst

חזרה על הסדר פעולות:



מבנה:



נכתוב את שלב 2:

מטרה: עושה פעולה תחבירית **בלבד**, מקבלת **ביטוי**, **שם** **וערך חדש** ומחליפה את כל הביטויים החופשיים של שם בערך החדש בביטוי, ואז נכתוב את eval מחדש בעזרת subst, היא **אינה** מבצעת הערכה.

חותמת של הפונקציה:

```
(: subst : WAE Symbol WAE -> WAE)
(define (subst expr from to) (todo) )
```

טסט:

```
(test (subst
       (Mul (Id 'x) (Id 'y)) ;; expr
        'x ;; from
        (Num 5) ;; to
        ) => (Mul (Num 5) (Id' y)))
```

מימוש:

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

נכתב ע"י צבי מינץ zvimints@gmail.com

כעת נעבור לפונקציית Eval:

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

שיעור 7:

מטרה: השפה שבינינו יודעת לחשב ביטויים אריתמטיים וגם יודעת לתת שמות מזהים לקטעי קוד וערכים, נרצה להוסיף יותר "**כוח**" לשפה ע"י מתן אפשריות להגדיר ביטוי שהוא **פונקצייה**.

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

זה מעין יצירת אובייקט חדש – עם פרמטר וגוף, רק נוכל להגדיר את הערך של x בהמשך ואז נעריך את הביטוי לפיו.

הערה: בשפת Racket ניתן לעשות את הפעולה הבאה:

```
> ( (lambda (x) (* x x)) 3 )
- : Integer [more precisely: Positive-Index]
9
```

בשפה שלנו נגדיר ככה:

```
> { call { fun { x } {* x x} } 3 }
```

היתרון הוא שנוכל להרחיב את הפונקציות בעזרת with באופן הבא:

```
{ with { pow2 {fun {x} {* x x}}} { + {call pow2 5} {call pow2 6} } }
```

לחלוקה של הקוד לפונקציות יש מספר **יתרונות**:

- פונקציות מאפשרות למתכנת להיות **מאורגן יותר.** בדרך כלל זה עוזר למקד את התוכנית על נושאים חשובים.
- פונקציות ממקדות פעולה אחת במקום אחד, כך שצריך להתרכז על המימוש וניפוי שגיאות רק פעם אחת.
- מפחית סיכוי לשגיאות אחרי שינויים, אם צריך לשנות משהו, **יש מקום אחד מרוכז**.
- פונקציות גורמות לכל התוכנית להיות קטנה יותר מכיוון שנעשה שימוש חוזר בחלקים של הקוד.
- פונקציות מאפשרות גם שימוש חוזר באותו הקוד בתוכניות אחרות, כך שהתוכניות נהיות מודולריות יותר. כתופעת לוואי נחמדה, השימוש בפונקציות בדרך כלל הופך את התוכנית ליותר קריאה.

הערות:

$$let\ ([x\ 5])\ (*\ x\ x)) \equiv ((lambda\ (x)(*\ x\ x))\ 5)$$

זה מתורגם אוטומטית ע"י Racket

הגדרות:

- פונקציות נותנות את היכולות לתת את החישוב עם **פרמטר אקטואלי אחר**

תקציר לקורס שפות תכנות

נכתב ע"י צבי מינץ zvimints@gmail.com

מבוסס על הרצאות של ד"ר עמרי ערן **ועל הסיכום שמופיע במודל של שנים קודמות**

- **מודל First Order:** שפונקציות הם לא ערך אמיתי, כלומר הם לא יכולים להיות משומשים או מוחזרים ע"י פונקציות אחרות, ולכן הם לא יכולים להישמר במבני נתונים, וזה בדרך כלל איך שפעם השתמשו בפונקציות בשפות תכנות.
- **מודל Higher Order:** פונקציות יכולות לקבל פונקציות אחרות כפרמטרים וגם להחזיר פונקציה כערך, זה מה שמקבלים עם C או עם Scala
- **First Class:** שפות בהם פונקציה היא טיפוס כמו כל טיפוס אחר, ואז ניתן לא לתת שם לפונקציה ואז זה מגדיל את היכולת של השפה, ואז ניתן להגדיר אותה בזמן ריצה, אפשר לחשוב על זה כמו שלמספר כלשהו Integer לא צריך לתת שם, לעומת מודל שבו כל Integer צריך לתת לו שם

כך זה נראה:

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

You have to do something like this:

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
```

בגלל שלא צריך לתת שם, אז אפשרות לעשות את הכל בשורה אחת, נותן **יכולת הבעה**

שינויים בקוד:

טסט:

```
(test (parse "{fun {x} {+ x x}}") => (Fun 'x (Add (Id 'x) (Id 'x))))
(test (parse "{call sqrt 5}") => (Call (Id 'sqrt) (Num 5)))
```

BNF:

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

  <num> - identifies any expression that pl evaluates to a Number
  <id> - identifies any expression that pl evaluate to a Symbol
  |#
```

Type:

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

[Div FLANG FLANG]
[With Symbol FLANG FLANG])

Parse-Sexpr:

```
(: parse-sexpr : Sexpr -> FLANG)
 (define (parse-sexpr expr)
(match expr
  [(number: num) (Num num)]
  [(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) (Div (parse-sexpr l) (parse-sexpr r))]
  [(list '* l r) (Mul (parse-sexpr l) (parse-sexpr r))]
  [(cons 'with _)
     (match expr
   [(list 'with (list (symbol: name) named-expr) body)
      (With name (parse-sexpr named-expr) (parse-sexpr body))]
   [_ (error 'parse-sexpr "bad with syntax in ~s" expr)])]
  [(cons 'fun more)
   (match expr
     [(list 'fun (list (symbol: parm-name)) body) (Fun parm-name (parse-sexpr body))]
     [else (error 'parse-sexpr "bad `fun` syntax in ~s" expr)]
  )]
  [(list 'call fun-expr arg-expr) (Call (parse-sexpr fun-expr) (parse-sexpr arg-expr))]
  [_ (error 'parse-sexpr "bad syntax in ~s" expr)]))
```

הערה:
ב-Call ובגוף של ה-With אז זה עם (Id 'x) בעוד שבצהרה של With ו-Fun אז זה Symbol, כל מה שעם **{}** **בלי אופרטור** נותן סימבול בעוד **שבלי** זה Id.
כלומר:

(test (parse "{call **f** 1}") => (Call (Id 'f) (Num 1)))
(test (parse "{fun **{ f }** { * **f f** }}") => (Fun 'f (Mul (Id 'f) (Id 'f))))
((((test (parse "{with **{f 1}** **{+ f f}**}") => (With 'f (Num 1) (Add (Id 'f) (Id 'f)

שיעור 8:
מטרה: לכתוב את פונקציית Eval

שיעור 9:

שיעור 10:

שיעור 11:

שיעור 12: