סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

שפות תכנות– הרצאה 1

מטלות: 5 מטלות, מהוות 20 אחוז מהציון הסופי.

מטרת הקורס: להבין את העקרונות שעומדים מאחורי בניית שפת תכנות.
לדוגמה: נהג מונית, הוא לא כמו כל אחד שצריך רק להניע את האוטו ולסוע. אלא בגלל שזה כלי העבודה שלו, רצוי שיכיר את כלי העבודה שלו לעומק. באותה מידה, כאשר אנו ניגשים לכתוב פרוייקט כלשהו, עלינו לדעת להתאים את בחירת השפה לאופי הפרוייקט.

בקורס זה נעבוד עם משפחת השפות Racket, השפות בה הן שפות פונקציונלית, נתמקד בשפת pl.

סקירת הקורס

- תוכנית כללית כיצד יתקיים הקורס:
  - נתכנן שפת תכנות.
  - נתחיל בשפה פשוטה מאוד–לשם ביצוע פעולות חשבון פשוטות.
  - עם הזמן נרחיב את השפה כדי לאפשר למתכנתים לבצע משימות מתוחכמות יותר.
- מטרת התהליך לעיל היא להביא את נקודת המבט של מעצבי שפות תכנות (תוך התחשבות בזו של מתכנתים).
  - האם כולנו צרכים להיות מסוגלים לעצב שפה? לא, אבל ברצוננו להבין מה עומד מאחרי החלטות רבות.
- מדוע עלינו לדאוג לשפות תכנות? כיום בכל פרוייקט כמעט יש שפה תכנות כלשהי.

מה הן היסודות של שפה?

- Syntax (תחביר)– הכללים הפורמליים לבניית תוכניות.
- Semantics (סמנטיקה)– המשמעות האמיתית של הדברים.

ההבדל בין תחביר לסמנטיקה: דרך טובה לחשוב על זה הוא ההבדל בין המחרוזת "42" (שני ערכי ASCII) המאוחסנת בקובץ איפשהו, לבין הספרה 42 המאוחסנת בזיכרון (בייצוג כלשהו). נוכל להמשיך עם הדוגמה שלמעלה: אין שוב דבר רע ב "שוד"– זו רק מילה, אבל שוד הוא משהו שניתן ללכת בעקבותיו לכלא.

מה יותר חשוב תחביר או סמנטיקה? סמנטיקה היא חשובה יותר, נמחיש זאת באמצעות דוגמה: נאתחל מערך כלשהו עם 15 תאים. כאשר נבקש בתוכנית לגשת לתא 25 בכל שפה התחביר הוא טיפה שונה, אבל נקבל את אותה תוצאה (פרט ל C) שהיא– לא ניתן לגשת לתא הזה, כי הוא לא אותחל בזיכרון ולכן הוא "לא קיים" בעצם. מכאן שהתחביר נועד לצורכי "קוסמטיקה" בלבד, וניתן לשנותו. וסמנטיקה היא הרבה יותר חשובה מתחביר, כי היא בעצם המשמעות.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

איך עלינו להתייחס לסמנטיקה?

ישנן מספר צורות ידועות לסמנטיקה. במהלך הניתוח נתעלם מנושאים פילוסופיים אפשריים (אך נהיה מודעים להם). למעשה ניתן לפתור אותם: לכל תוכנית יש הסבר רשמי, כלומר ניתן לקחת תוכנית שאנו כותבים ולהמיר אותה ללוגיקה. אנחנו נשתמש ב Racket מכמה סיבות: תחביר, פונקציונליות, פרקטי, פשוט, רשמי, הקלדה סטטית, סביבה.

מבוא ל Racket

- שפת Racket שייכת למשפחת ה Lisp(=משפחה של שפות תכנות פונקציונליות).
- התחביר של שפת Racket דומה לשפות מבוססות S-expression.
- ליבת השפה ממומשת בשפת C.
- בשפה טיפוסים דינמיים. כלומר, לא ניתן לדעת מראש איזה טיפוס הפונקציה מקבלת ומחזירה.
- תיעוד: לשפת Racket תיעוד נוח וידידותי מאוד.

The evaluation function שמשתמשים בה ב Racket היא למעשה פונקציה שלוקחת חתיכת תחביר ומחזירה (או מבצעת) את הסמנטיקה שלה.

הערה צדדית (שפות תכנות הן חיות ומשתנות):

E.W. DIJKSTRA

"Goto Statement Considered Harmful."

המאמר הזה הוא של החוקר הידוע המנסה לשכנע אותנו כי יש לבטל את הצהרת ה GOTO הידועה משפות התכנות שלנו (אסמבלי).

תחביר השפה

קבצי Typed Racket נפתחים בשורה:

```
# lang typed/racket
; ; Program goes here.
```

אבל אנחנו אנחנו נשתמש בגרסה של שפת Racket שפת PL ולכן נפתח עם השורה:

```
# lang pl
; ; Program goes here.
```

עצמים בסיסיים בשפת Racket:

- שני ערכים בוליאניים מובנים: true/false

```
# t        ; another name for true , and the way it prints
# f        ; ditto for false
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

- מספרים

```
1
0.5
1/2        ; this is a literal fraction , not a division operation
1+ 2 i     ; complex number
```

- Symbols

```
' apple
' banana-cream-pie
' a->b
' #% $ ^ @*& ?!
; (: print-type ' apple )
```

- מחרוזות Strings

```
" apple "
" banana cream pie "
```

- תווים-Chatacters

```
#\ a
#\ b
#\ A
#\ space  ; same as #\( string #\ a #\ b )
( with a space after \)
```

Prefix Expressions

```
( not true )        ; => #f
(+ 1 2)             ; => 3
(< 2 1)             ; => #f
(= 1 1)             ; => #t
( string-append " a " " b " )        ; => " ab "
( string-ref " apple " 0)            ; => #\ a
```

```
(eq? 'apple 'apple)        ; Object identity
(eq? 'apple 'orange)       ; => #f
(eq? "apple" "apple")      ; => depends

(equal? "apple" "apple")   ; => Content equality
(string=? "apple" "apple"); => ... for strings

(null? null)              ; => #t

(number? null)             ; => #f
(number? 12)               ; => #t
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

הערות

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

;; A common convention is to use two
;; semi-colons for multiple lines of
;; comments, and a single semi-colon when
;; adding a comment on the same line as
;; code.

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

שפות תכנות – הרצאה 2

ביטויים ב Racket

- עצמים: כמו מחרוזות, מספרים וכדומה. נכתבים ללא סוגריים.
- ביטויים הכוללים אופרטורים (נכתבים עם סוגריים).
- ביטויים מיוחדים המשתמשים במילות שמורות כמו: התניות, השמות, הכרזה וכדומה (נכתבים עם סוגריים).

דוגמה לביטוי עם אופרטורים:

(+ 2 (* 3 4)) (– (+ 1 2) 3)

איך Racket מפרש ביטויים?

- הביטוי מתפרש מצד שמאל לימין.
- כל פעם נכנסים על לסוגריים הכי פנימיות.

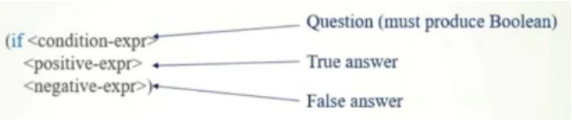
(+ 2 12 (– (+ 1 2) 3))
(+ 2 12 (– 3 3)
(+ 2 12 0)
output: 14

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

התניות



דוגמה:

(if (< 2 3) 10 30)

output: 10

איך מפרשים זאת? האם 2 קטן מ-3 אם כן פלוט 10 אחרת פלוט 30.
2 אכן קטן מ-3 ולכן הפלט הוא 10.

ב Racket כל ערך חוץ מ #f נחשב לערך אמת. ולכן הפלט של התנאי הבא יהיה 3:

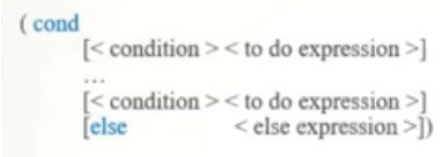
(if 10 3 0)

שפת Racket היא שפת תכנות פונקציונאלית, ולכן לכל חישוב שנעשה יש ערך מוחזר.
לכן, אם נסתכל על הדוגמה:

if(test do_somthing)

אם הערך של test הוא אמת אזי יוחזר do_somthing. אבל מה יקרה אם test הוא שיקרי?
אם test הוא שקרי, לכאורה לא צריך לקרות שום דבר. אולם, מכיוון שהשפה היא שפה פונקציונלית, גם במקרה שהערך הוא שקרי, חייב להחזיר ערך כלשהו. לכן, Racket הגדירה מקרה זה כטעות תחבירית.

דרך נוספת לכתוב התניות היא cond. Cond מאפשר לנו לבדוק מספר התניות כמו if else if.
ברגע שאחד התנאים מתקיים הוא מבצע אותו ולא ממשיך לשאר התנאים.



דוגמה:

```
(cond
    [(eq? 'a 'b) 0]
    [(eq? 'a 'c) 0]
    [else 2])
```

output: 2

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

תופעות לוואי side effects

תופעות לוואי זה דבר מסויים שקורה למערכת בזמן שעובדים עליה. מבחינת מימוש השפה (הקוד) אין שום הבדל, המתכנת לא רואה הבדל.
בשפת racket אין תופעות לוואי.
בשפת Racket כמעט לכל דבר יש ערך מוחזר. יוצאי דופן הם: if ו define –להם אין ערך מוחזר.

רשימות ורקורסיות

רשימה הינה טיפוס בסיסי ב racket. רשימה הינה אוסף של איברים בתוך סוגריים עגולים. רשימה יכולה להיות ריקה.
נגדיר רשימה באופן רקורסיבי:

- הרשימה הריקה, זהו מקרה הבסיס. סימונים: '() , null .
- זוג (pair) שהאיבר הראשון שלו יכול להיות כל דבר, והאיבר השני שלו הוא רשימה.
- Pair מיוצר על ידי הפרוצדורה cons. האיבר הראשון מתווסף לראש הרשימה ונוצרת רשימה חדשה. במילים אחרות cons מכניס איבר לראש רשימה קיימת.
- ניתן להשתמש בפונקציה list על מנת לייצג רשימה.
- ניתן לייצג רשימה על ידי סוגריים וגרש: '(1).

דוגמאות:

```
null                          ; => '()

(list 1 2 3)                   ; => '(1 2 3)
(cons 0 (list 1 2 3))          ; => '(0 1 2 3)

(cons 1 null)                 ; => '(1)
(cons 1 '())                   ; => '(1)
(cons 'a (cons 2 null))       ; => '(a 2)

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

לשם בדיקה האם טיפוס האובייקט הוא מסוג list ניתן להשתמש ב list?. לבדיקת האם אובייקט מסוים הוא pair נשתמש ב pair?.

(list? '(1 2)) → true
(list? (cons 1 2)) → false
(pair? (cons 1 2)) → true

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

Functions on lists

append: איחוד מספר רשימות סופיות לרשימה אחת. Append בעצם לוקח מספר נתון של רשימות סופיות, פותח אותן, ושם את איבריהן לפי סדר הופעתן, ברשימה אחת חדשה וסוגר אותה.
הערה: append של רשימה כלשהי עם הרשימה הריקה יחזיר את אותה רשימה.

```
(append (list 1 2) null) ; => '(1 2)
(append (list 1 2)
        (list 3 4))      ; => '(1 2 3 4)
(append (list 1 2)
        (list 'a 'b)
        (list true))     ; => '(1 2 a b #t)

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

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

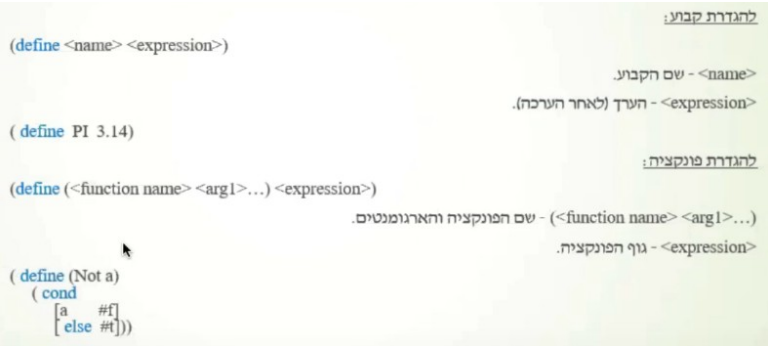
פונקציות גישה לאיברים ברשימה:

- first: מחזיר את האיבר הראשון ברשימה.
- Rest: מחזיר את אברי הרשימה, פרט לאיבר הראשון.
- List–ref: לגשת לאינדקס מסויים ברשימה.

Define and functions

נשתמש ב define בשביל ההגדרות הבאות:

- הגדרה של קבוע.
- הגדרה של פונקציה: כאשר מגדירים פונקציה יש להגדיר מה סוג הקלט ומה סוג הפלט. כלומר, איזה סוג של משתנים הפונקציה יכולה לקבל, ואיזה סוג של משתנים הפונקציה מחזירה.



סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

הצהרה של פונקציה

הצהרה של פונקציה תראה כך:

(:<name of function>: <input type> → <output type>)

הצהרת הפונקציה, סוג הקלט וסוג הפלט:

```
(: f : Number -> Number)
```

הגדרת הפונקציה:

```
(define (f x)
    (if (and (>= x 0)(<= x 5)) 1 0))
```

ביצוע בדיקות:

```
(test (f 0) => 1)
(test (f 6) => 0)
```

רקורסיות

```
(: list-length : (Listof Any) -> Natural)
(define (list-length list)
  (if (null? list)
    0
    (+ 1 (list-length (rest list)))))
```

הסבר: זוהי פונקציה הנקראת list-length המקבלת רשימה של כל דבר, ומחזירה מפסר טבעי (אורך הרשימה). תחילה נגדיר את תנאי העצירה- האם הרשימה ריקה. אם הרשימה ריקה, נחזיר 0. אחרת נוסיף 1 לתוצאה ונקרא לפונקציה שוב ללא האיבר הראשון. מכיוון שזוהי פונקציה רקורסיבית נגיע בסופו של דבר לרשימה ריקה, ואז נחזור אחורה במחסנית הקריאות על להתחלה. ולבסוף נחזיר את התוצאה שהיא אורך הרשימה. אולם, יש כאן בעיה רצינית- הרקורסיה הנ"ל יכולה לגרום למצב של stack overflow. איך נפתור בעיה זאת? באמצעות רקורסיית זנב!

רקורסיית זנב

נשתמש ברקורסיית זנב על מנת שלא נצטרך לשמור את כל הקריאות במחסנית, ולחזור תמיד אחורה. ברקורסיית זנב כשנגיע לסוף נדע את התוצאה, ללא צורך בחזרה לאחור במחסנית הקריאות.

תחילה נראה חישוב עצרת באמצעות רקורסיה רגילה:

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

השלבים לחישוב בעזרת פונקציה רקורסיבית:

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

כעת נראה חישוב עצרת באמצעות רקורסיית זנב, פונקציית זנב בדרך כלל דורשת פונקציית עזר:

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

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

השלבים לחישוב באמצעות רקורסיית זנב:

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

שימוש ברקורסיית זנב אינו דורש שמירת הקריאות במחסנית, אין בה נסיגה לאחור, ולכן לא יכול להיווצר מצב של הצפת זיכרון. לכן נעדיף להשתמש ברקורסיית זנב.

הגדרת טיפוסים

הסיבות שנצטרך טיפוסים סטטיים:

- נרצה שיהיה סדר, ומבניות בקוד.
- נרצה להקטין את מספר הבאגים, יכול לסייע במציאת שגיאות.
- שיפור זמן ריצה.

באמצעות הפקודה define-type ניתן להגדיר טיפוס חדש.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

לדוגמה: נגדיר טיפוס חדש בשם Bool שיכול להיות 0 או 1, בצורה הבאה:

```
(define-type Bool = (U 0 1))
```

הסימן U מייצג איחוד של n איברים שנציין אחריו.

ישנה אפשרות להגדיר טיפוס המכיל מספר אפשרויות (בנאים).

לדוגמה: נגדיר טיפוס חדש בשם Animal:

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

באופן זה הגדרנו בנאים לטיפוס החדש Animal:

- הבנאי Snake מקבל סימן, מספר, וסימן ומחזיר Animal.
- הבנאי Tiger מקבל סימן ומספר, ומחזיר Animal.

ובאופן כללי הגדרת טיפוס תיראה כך:

```
(define-type <name>
    [<variant1> <type> ...]
    [<variant2> <type> ...]
    [...])
```

כאשר name זה שם הטיפוס החדש. וכל variant הוא שם של בנאי עם הטיפוסים המתאימים.

לאחר הכרזה על טיפוס מסויים Racket מגדירה מספר פונקציות מובנות עבור הטיפוס.

לדוגמה: לאחר ההכרזה על Animal נוכל להשתמש בפונקציה Animal ?

```
(Animal? ( Tiger 'Tony 12)
#t
(Animal? ( Snake 'Slimey 10 'rats)
#t
(Animal? ( cons 'Robi 19)
#f
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

הפונקציה cases

cases היא פונקציה המיועדת עבור טיפוסים שהם user-defined, כלומר טיפוסים שהמתכנת יוצר.
הפונקציה מאפשר לבדוק עבור משתנה מסוג הטיפוס את ה variant המתאים לו על ידי תבניות.
לדוגמה:

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

הפונקציה cases מופעלת על אובייקט X מסויים, במקרה שלנו הוא מטיפוס Animal. ל Animal יש שני בנאים Snake ו Tiger, חובה לבדוק כל אחד מהבנאים. במידה ו X הוא Snake תתבצע וריפיקציה, ועכשיו n יהיה שווה לסימבול של Snake .

ב cases יש לכסות את כל ה variants של הטיפוס, אחרת נקבל שגיאה.
ב cases אין צורך ב else מכיוון שאם קיבלנו משתנה מסוג הטיפוס, הוא חייב להיות מתאים לתבנית של אחד מה variants.
במידה ולא נרצה לציין את כך הבנאים, ניתן להשתמש ב else בסוף, אך בדרך כלל נעדיף שלא.

דקדוק חסר הקשר BNF

חוזרים לנושא הקורס: אנו רוצים לחקור שפות תכנות, ואנחנו רוצים לעשות זאת "בעזרת" שפת תכנות. הדבר הראשון שאנחנו רוצים לעשות כאשר אנו מעצבים שפה הוא לפרש את השפה.
לשם כך אנו משתמשים ב- BNF, דקדוק חסר הקשר הוא מודל חישוב השקול לאוטומט מחסנית.
נרצה להגדיר מה חוקי ומה לא חוקי בשפה. לשם כך נשתמש ב BNF.
לדוגמה, הנה ההגדרה של שפה אריתמטית פשוטה:

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

נתחיל עם <AE>, שאמור להיות אחד מאלה:

- מספר <num>
- <AE>, הטקסט " +", ו <AE> אחר
- אותו דבר אבל עם "-"

צריך להיות ברור שה- " +" וה- "-" הם דברים שאנו מצפים למצוא בקלט - מכיוון שהם לא עטופים ב <>. כל אחד מהם הוא טרמינל.
<num> הוא גם טרמינל: כשנגיע אליו בגזרה, סיימנו. (אותיות קטנות)
<AE> הוא לא טרמינל: כשאנחנו מגיעים אליו, עלינו להמשיך באחת האפשרויות.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

לדוגמה, אנחנו יכולים לקבל את הביטוי "1-2+ 3" באמצעות סדרת הגזירה הבאה:

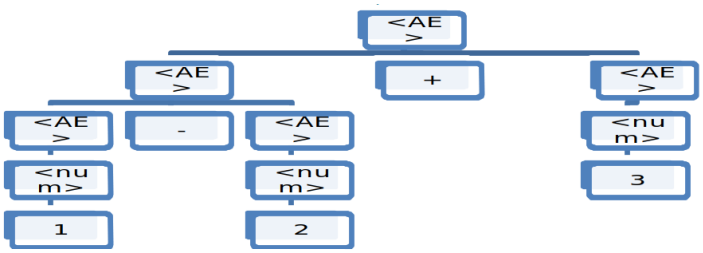
```
<AE>                      ;          ==>
  <AE> + <AE>             ; (2)      ==>
  <AE> + <num>            ; (1)      ==>
  <AE> - <AE> + <num>     ; (3)      ==>
  <AE> - <AE> + 3         ; (num)    ==>
  <num> - <AE> + 3        ; (1)      ==>
  <num> - <num> + 3       ; (1)      ==>
  1 - <num>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

בעקרון היינו אמורים לציין את <num> כ- <NUM> (כלל גזירה- לא טרמינל) ולהגדיר את הכלל:

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

אבל אנחנו לא - מדוע?
מכיוון שב Racket יש לנו מספרים פרימיטיביים ואנחנו רוצים להשתמש ב Racket כדי ליישם את השפה שלנו. זה מקל עלינו, אנחנו מקבלים דברים בחינם כמו rationals, floats, וכו'.

ראינו דרך אחת לייצוג סדרת גזירות, אנו יכולים להציג גזירות גם באמצעות עץ, עם הכללים שהשתמשנו בהם בכל צומת. זה נקרא עצי גזירה והם נראים כך:



עצי גזירה מגדירים מילה על ידי שרשור המילים שבעלי העץ משמאל לימין.

דו-משמעות

דו-משמעות: ביטוי יכול להגזר במספר דרכים (כלומר, יש מספר עצי גזירה). ניתן לגזור מספר "123" בשתי דרכים המביאות לעצים שנראים שונים. דו-משמעות זו אינה בעיה "אמיתית" כעת, אך היא תהפוך לבעיה בקרוב מאוד. אנו רוצים להיפטר ממנה, נרצה שתהיה דרך אחת לגזור כל ביטוי.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

ישנם מספר דרכים להיפטר מדו משמעות כמו:

```
Example 1 - Left association:
Instead of allowing an <AE> on both sides of the operation,
we force the left one to be a number:

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

Now there is a single way to derive any expression, and it
is always associating operations to the right: an expression
like "1+2+3" can only be derived as "1+(2+3)".

Example 2 - Right association:
To change this to left-association, we would use this:

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

Example 3 - Semantically required precedence:
But what if we want to force precedence?  Say that our AE
syntax has addition and multiplication (now (1+2)*3 is no
longer semantically the same as 1+(2*3)):

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

Loosing Ambiguity – the simpler solution

האם יש פתרון יותר טוב?

זה ממש מולנו: Racket עושה את זה– נשתמש תמיד בביטויים עם סוגריים:

```
<AE> ::= <num>
       | ( <AE> + <AE> )
       | ( <AE> – <AE> )
```

כדי למנוע בלבול בין קוד Racket לקוד בשפה שלנו, אנו גם משנים את הסוגריים לסוגריים מתולתלים:

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

Using Prefix notation

נאמץ את פתרון של Racket ונשתמש prefix notation עם סוגריים מתולתלים:

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

שפות תכנות – הרצאה 3

המטרה שלנו: לבנות parser פשוט ב pl עבור הביטוי:

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

Implementing a "parser"

בלי קשר לאיך נראה התחביר, אנו רוצים לנתח אותו בהקדם האפשרי – המרת ה- concrete syntax ל-abstract syntax tree.

לא משנה כיצד אנו כותבים את התחביר שלנו:

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

לכל הדוגמאות הללו אותו עץ מופשט- מפעיל " +" עם שני אופרנדים 3 ו 4. (סמנטית, אנו מתכוונים לאותו הדבר- הוספת המספר 3 והמספר 4). התחביר המופשט הוא בעצם מבנה עץ עם פעולת פלוס כשורש ושני עלים המחזיקים את שני המספרים.

אנו יכולים לתאר זאת ב Racket כביטוי ((Add (Num 3) (Num 4) שבהם 'Add' ו- 'Num' הם בנאים של עץ התחביר.

נגדיר טיפוס חדש AE המייצג את העץ הבא:

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

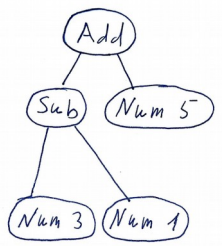
לדוגמה, בהינתן הביטוי הבא:

(+ (- 3 1) 5)

הקוד יראה כך:

(Add (Sub (Num 3) (Num 1)) (Num 5))

והעץ יראה כך:



סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

S-expression

כל מספר או סימבול הוא s-expr. רשימה של s-expr היא s-expr. בראקט קיימת פונקציה שמקבלת מחרוזת, וממירה אותה לביטוי s-expr. ניתן לייצר ביטוי s-expr באמצעות עץ.

הפונקציה let

באמצעות define אנו מגדירים משתנה גלובלי. ניתן גם להגדיר משתנה לוקאלי, נעשה זאת באמצעות הפקודה let. מטרתה היא ביצוע קישור בין שמשתנים. בלוק let יראה כך:

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

דוגמת שימוש:

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

הפונקציה match

```
( match value
    [ pattern1  result-expr1 ]
    [ pattern2  result-expr2 ]
    …)
```

הפונקציה match מחזירה את הערך (result-expr) לתבנית הראשונה שמתאימה.
תבנית שתמיד מצליחה:

```
( match ( list 1 2 3) [x x])
'(1 2 3)
```

ניתן להשתמש בערכים שלהם ביצענו קישור בתבנית, לדוגמה:

```
( match '(1 2 3)
    [( list x y z)  (+ x y z)])
6
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

הסבר: הפונקציה רואה שהתבנית היא (list x y z) אכן מתאימה ל value שהוא כאמור (list 1 2 3). ולכן הפונקציה מחזירה את מה שמופיע ב result-expr שזה חיבור שלושת האיברים:
6= 1+ 2+ 3

בחזרה לבניית ה parser שלנו, המטרה שלנו היא להפוך מחרוזת ל S-expr. s-expr מוגדר כביטוי סופי או ביטוי מהצורה של זוג סדור, כאשר כל אחד מאיברי הזוג הוא גם S-expression. s-expr הוא עץ בינארי.

נגדיר טיפוס חדש AE המייצג את העץ הבא:

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

ונגדיר את ה parser שלנו כך:

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

הפונקציה מקבלת מחרוזת וממירה אותו לטיפוס AE, היא עושה זאת על ידי קריאה לפונקציית עזר parse-sexpr.

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

Parse-sexpr מקבלת ביטוי מסוג sexpr וממירה אותו לטיפוס מסוג AE- אותו מבנה נתונים המייצג עץ שקל לנו לעבוד איתו.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

שפות תכנות – הרצאה 4

היום נעבוד על הסמנטיקה, evaluetion. מאפיין חשוב של ה BNF הוא: אנחנו יכולים להשתמש בסדרת הגזירה כדי לציין משמעות. המשמעות בהקשר שלנו היא "להפעיל" את התוכנית (או "compiling", "interpreting", אבל אנחנו נשתמש ב "evalueting").

```
a. eval(<num>) = <num>                    ; <-- special rule: moves syntax into a value
b. eval(<AE1> + <AE2>) = eval(<AE1>) + eval(<AE2>)
c. eval(<AE1> - <AE2>) = eval(<AE1>) - eval(<AE2>)
```

נשים לב שיש לנו שוב בעיית דו- משמעות:

```
eval(1 - 2 + 3) = ?
```

ניתן לפרש ביטוי זה בשתי דרכים:

```
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
```

אנחנו רואים מדוגמה זו שב eval דו-משמעות היא בעייתית מאוד! לכן אסור לנו לאפשר לגזור תחביר כלשהו בכמה דרכים, אחרת eval שלנו תהיה לא-דטרמיניסטית.

Compositionality

מאפיין חשוב בתחביר, קומפוזיציות: משמעות הביטוי המורכב נקבעת על ידי מבנהו ומשמעויות מרכיביו.

Implementing an Evaluator

כעת נמשיך ליישם את הסמנטיקה של התחביר שלנו, אנו מבטאים זאת באמצעות פונקציית `eval' המעריכה ביטוי.

אנו משתמשים בעקרון תכנות בסיסי, פיצול הקוד לשתי שכבות, אחת ביצוע parsing על הקלט והשניה היא ביצוע evaluation. פעולה זו מונעת בלאגן, ויותר נכונה מכיון שהתחביר והסמנטיקה הם חלקים שונים קונספטואלית. יתרון נוסף הוא שעל ידי שימוש בשני רכיבים נפרדים, פשוט להחליף כל אחד מהם, מה שמאפשר לשנות את תחביר הקלט, ואת הסמנטיקה באופן עצמאי.

וכעת המימוש:

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

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

כעת מה שנותר לעשות הוא לשלב בין parse שכתבנו כבר לבין eval שכתבנו הרגע, נבצע שילוב זה באמצעות פונקציית run בודדת שעושה evalueting למחרוזת AE.

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

למעשה כעת הפונקציה run נהייתה נקודת הממשק היחיד של המשתמש עם הקוד שלנו,
והפונקציה היחידה שיש להשתמש בבדיקות המאמתות את הממשק שלנו:

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

וכעת הקוד המלא:

```
#lang pl

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

;; AE abstract syntax trees
(define-type AE
 [Num Number]
 [Add AE AE]
 [Sub AE AE]
```

```
  [Mul AE AE]
  [Div AE AE]  )

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

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

(: eval : AE -> Number)
;; consumes an AE and computes the corresponding 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))]))

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

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

שפות תכנות – הרצאה 5

Binding and Subsitutions

Substitution

מושג חושב מאוד בשפות תכנות. אפילו בשפה הפשוטה שלנו, אנחנו נתקלים בביטוים שחוזרים על עצמם. לדוגמה:

```
{* {+ 4 2} {+ 4 2}}
```

למה חשוב לנו, להפתר מהחזרתיות הזאת?

- חישוב מיותר: לדוגמה בביוטוי שלמעלה אנחנו מחשבים פעמיים את אותו הביטוי, וזה מיותר.
- חישוב יותר מסובך: החישוב יכול להיות פשוט יותר לולא חזרתיות זו.
- שכפול קוד הוא דבר רע: שכפול קוד יכול להוביל לבאגים שהיו יכולים להנמע אילולא היינו עושים זאת. נניח וברצוננו לשנות את הקוד מסיבה כלשהי, אנו יכולים לשנות את הקוד במקום אחד ולשכוח לשנות את המקום השני.
- הוספת יכולת הבעה למתכנת: אנחנו לא רק אומרים שאנו רוצים להכפיל שני ביטויים ספציפיים כלשהם. אלא אנחנו אומרים שאחנו מכפילים ביטוי כלשהו בעצמו.

נרצה להוסיף לשפה שלנו משתנים, כמו לדוגמה:

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

מזהים

הדרך הרגילה להימנע מכפל קוד הוא להוסיף משתנים. על מנת להוסיף לשפה שלנו את היכולת להגדיר משתנים נוסיף את הביטוי with לשפה שלנו:

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

כאשר:

באדום– שם המשתנה שאותו נרצה להגדיר.

בכחול– הביטוי שמקבל את השם.

בירוק– הגוף, בו נעשה שימוש בשם.

זה בעצם כמו let, אבל מכיוון שאנו כותבים שפה חדשה, נשתמש ב with במקום ב let.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

הוספת מזהים לשפה שלנו AE: שפה חדשה WAE

נוסיף את האופציה להגדרת מזהים לשפה שלנו, נתחיל עם ה BNF.

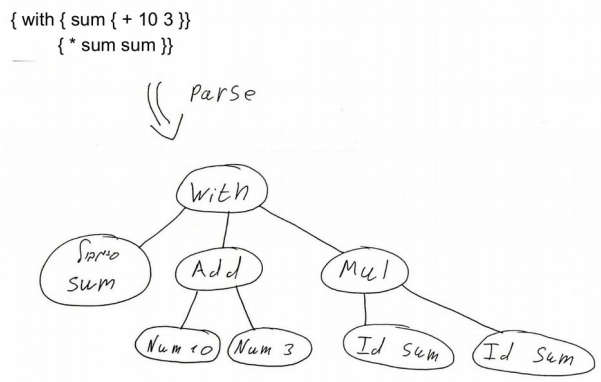
```
<WAE> ::= <num>
        | { + <WAE> <WAE> }
        | { - <WAE> <WAE> }
        | { * <WAE> <WAE> }
        | { / <WAE> <WAE> }
        | { with { <id> <WAE> } <WAE> }
        | <id>
```

נשים לב כי הוספנו לשפה שני חוקים חדשים:

- הראשון, כדי שנוכל להכניס מזהה.
- השני, לשם השימוש בו.

<id> מייצג כל symbol אפשרי. הוא מעין קבוע שלא ניתן לשנות את ערכו. כלומר, הוא ישמש אותנו כשם מזהה לקטע קוד.

לדוגמה:



נגדיר את הטיפוס החדש אצלנו בשפה כך:

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

סיכום קורס שפות תכנות

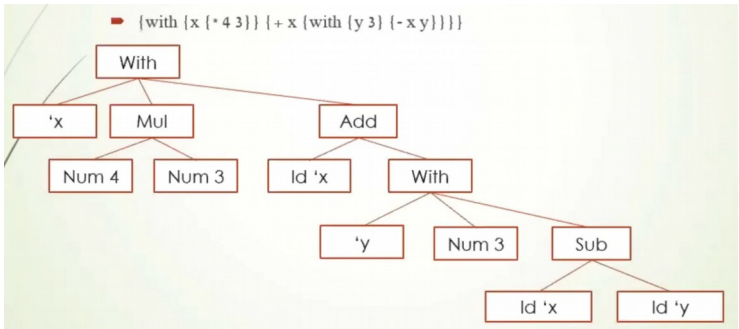
נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

נרחיב גם את הפרסר שלנו על מנת שידע לקבל את התוספת:

```
(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     ;; go in here for all sexpr that begin with a 'with
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad ‘with' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

דוגמה נוספת:



<u>שפות תכנות – הרצאה 6</u>

ראינו כי ניתן להגדיר מבנה המאפשר להכריז על שם מזהה חדש, לתת לו ערך ולהשתמש בו. לאחר מכן, הרחבנו את השפה וקראנו לה WAE. הרחבנו את פונקציית ה parse שבהינתן ביטוי מהצורה with היא תחזיר עץ אבסטרקטי.

מימוש evaluation of ‘with’: subtitutions

כדי לגרום לזה לעבוד, אנחנו צרכים משהו שיבצע החלפות. אנחנו רוצים להעריך את הביטוי:

```
{with {id WAE1} WAE2}
```

אנחנו צרכים להעריך את WAE2 עם id שהוא מוחלף על ידי WAE1. באופן פורמאלי:

```
eval( {with {id WAE1} WAE2} ) = eval( WAE2[eval(WAE1)/id] )
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

הערכה של ביטוי with:

1. הערך את WAE1 וקבל בחזרה ערך v.
2. החלף את כל המופעים התמאימים של id ב WAE2 בערך v.
3. הערך את WAE2.

נתמקד בשלב 2, ביצוע ההחלפות:
איך נחליף את כל המופעים "המתאימים" של name בתוך ה body בערך v?
נגדיר פונקציה שמקבלת את תת העץ של body, וגם symbol של ה name וערך v. מה שהפונקציה תעשה זה לחפש את כל המופעים של ה name ב body ולהחליף אותם ב v.

נסיון ראשון:
נסמן E[v/i] – רוצים להחליף את המופעים של i בערך V בתוך הביטוי E.
נסיון זה עובד עבור דוגמאות פשוטות כמו:

```
{with {x 5} {+ x x}}   -->  {+ 5 5}
{with {x 5} {+ 10 4}}  -->  {+ 10 4}
```

הבעיה: אם נפעיל with מקונן, זה כבר לא יעבוד:

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

לא היינו אמורים להחליף את ה X הפנימי כי הוא הוא מייצג את ה symbol הפנימי. התפקיד של ה X הפנימי הוא לתת שם למזהה אחר, ולא היינו אמורים להחליפו בערך.

מושגים:

- מופע של מזהה נקרא: binding instance, משמש להצהרה על שם חדש.
- Scope: האזור בקוד ביחס ל binding instance, שהמופעים של אותו שם קשורים אליו.
- Bound instance: מופע של I שאינו binding ונמצא ב scope של מופע binding של I.
- Free instance: מופע שאינו binding ואינו bound.

נסיון שני:
נסמן E[v/i] – נרצה להחליף מופעים של I בערך V בתוך הביטוי E שאינם binding instance.
האומנם נצליח לסדר את הבעיה שנתקלנו בה בנסיון הקודם:

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

אבל כאשר ננסה להרחיב את הביטוי נקבל שוב שגיאה:

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

קיבלנו שהתוצאה היא 10, אבל ציפינו לקבל 8. וזה מאחר שה X הפנימי צריך להיות מקושר ל X שהכי קרוב אליו. צריך לדעת לשייך כל משתנה ל scope המתאים.

נסיון שלישי:

נסמן E[v/i] – נרצה להחליף מופעים של i בערך V בתוך הביטוי E שאינם binding instance, ואינם בתוך scope של מזהה אחר.

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

יתורגם ל:

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

וזו שגיאה מכיוון ש–X unbound (ואין כלל סביר שנוכל לציין כדי להעריך אותו).

הבעיה: שהמנגנון החלפה שלנו נעצר בכל scope חדש, במקרה זה הוא נעצר ב scope שבו y מוגדר ללא צורך וזה מכיין שב scope זה מוגדר שם אחר.

נסיון רביעי:

נסמן E[v/i] – החלף את כל המופעים של i בתוך הביטוי E שהם free instances בערך V.

סוף סוף, זוהי הגדרה טובה.

נותר רק לכתוב את הפונקציה subst שמבצעת את ההחלפה:

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

יש לנו עדיין באג בפונקציה הנ"ל נטפל בזה בהמשך.

כעת נגדיר נותר להגדיר את הפונקציה eval שתבצע הערכה על הביטוי.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

שפות תכנות – הרצאה 7

מתי ואיך נשתמש בפונקציה subst בפונקציית ההערכה

אנחנו נצטרך להגדיר חוקים חדשים שידעו להתמודד עם קטעי הקוד שיש בהם "with".
נניח וניתן לנו ערך מהצורה:

```
{with {x E1} E2}
```

נצטרך:

1. להעריך את E1 על מנת לקבל את הערך V.
2. נרצה להחליף את המופעים את המזהה X בערך V בביטוי E2.
3. נבצע הערכה על הביטוי המתקבל לאחר ההחלפה.

במילים אחרות, נגדיר את הערכה שלנו כך:

```
eval( {with {x E1} E2} ) = eval( E2[eval(E1)/x] )
```

אז כעת אנחנו יודעים כיצד להעריך ביטויי with.

Evaluating identifiers

המאפיין העיקרי של subst, הוא שהוא לא משאיר לא free instances של המשתנה בסביבה.
זה אומר שאם הביטוי תקין (לא מכיל מזהה חופשי), אז כאשר אנחנו עוברים מ

```
{with {x E1} E2}
```

ל

```
E2[E1/x]
```

בתוצאה שנקבל אין לא free instances של X. אז אנחנו לא צרכים לטפל במזהים בפונקציית ההערכה שלנו, כי הם כבר מטופלים בפונקציה subst. ואם נתקל במזהה אזי מדובר בשגיאה כי הוא חופשי.
אז נוסיף זאת להגדרה הפורמאלית שלנו של WAE:

```
eval(...) = ... same as the AE rules ...
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(id) = error!
```

הערה: נשים לב כי הפונקציה subst צורכת WAE. אולם, מה שאנחנו מקבלים כאשר אנחנו עושים eval(E1) זה מספר. ניתן לפתור זאת ע"י עטיפת המספר על ידי בנאי Num.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

המימוש:

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

נחבר את הכל יחד ונקבל את הקוד הבא, אבל עם הרצת הקוד נגלה בעיה בלתי צפויה:

```
#lang pl

 #| BNF for the WAE language:
      <WAE> ::= <num>
              | { + <WAE> <WAE> }
              | { - <WAE> <WAE> }
              | { * <WAE> <WAE> }
              | { / <WAE> <WAE> }
              | { with { <id> <WAE> } <WAE> }
              | <id>
 |#

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

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

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

(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (if (eq? bound-id from)
       expr
       (With bound-id
             named-expr
             (subst bound-body from to)))]))

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

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

;; tests
(test (run "5") => 5)
(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}} {+ y y}}}") => 14)
;; in reality returns -- eval: free identifier: x
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
(test (run "{with {x 5} {with {x x} x}}") => 5)
```

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

ישנה עדיין בעיה, הבעיה היא עם ביטויים כמו:

```
{with {x 5}
    {with {y x}
      y}}
```

השגיאה היא ש X מזוהה כמזהה חופשי.
אבל הוא היה אמור להיות מוערך לערך 5 ...
הבעיה: אנחנו לא נכנסים ל with הפנימי כשאנחנו מחליפים 5 עבור ה X החיצוני. זאת מאחר שיש לו את אותו השם X, אבל אנחנו צרכים להכנס לתוך ה named expression שלו.
עלינו לבצע החלפה ל named expression גם אם המזהה יש לו את אותו השם של המזהה שאותו אנו מחליפים:

```
(: subst : WAE Symbol WAE -> WAE)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (define (subst expr from to)
    (cases expr
      [(Num n) expr]
      [(Add l r) (Add (subst l from to) (subst r from to))]
      [(Sub l r) (Sub (subst l from to) (subst r from to))]
      [(Mul l r) (Mul (subst l from to) (subst r from to))]
      [(Div l r) (Div (subst l from to) (subst r from to))]
      [(Id name) (if (eq? name from) to expr)]
      [(With bound-id named-expr bound-body)
       (With bound-id
             (subst named-expr from to)
             (if (eq? bound-id from) ;; new - only ask on body
               bound-body
               (subst bound-body from to)))]))
```

כעת אם נריץ את הבדיקות על הקוד הנ"ל הם יעברו ללא שגיאות.

שפות תכנות – הרצאה 8

עכשיו כשיש לנו תבנית עבור משתנים מקומיים, שאילץ אותנו להתמודד עם with והחלפות הולמות וכל מה שקשור בזה. כעת השפה שבנינו יודעת לחשב ביטויים אריתמטיים, וגם יודעת לתת שמות מזהים לקטעי קוד וערכים. אך נראה שעדיין אין לה מספיק כח, נרצה לפתח את השפה שלנו עוד, נרצה להוסיף פונקציות לשפה שלנו. נרצה להגדיר ביטוי עם משתנה כלשהו עם מעין "הבטחה לחישוב" עבורו. לדוגמה, נניח ונתון ביטוי ה with הבא:

```
{with {x 5}
    {* x x}}
```

כאן, ה X כפול X הוא הגוף פרמטר בפני עצמו עבור ערך כלשהו X. אם ניקח את הביטוי הזה ונוציא את 5, יש פה את כל המרכיבים הדרושים לפונקציה:

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

```
{with {x}
    {* x x}}
```

A new form for anonymous functions

כל מה שעלינו לעשות זה להחליף את ה with בביטוי הקודם בשם מתאים אחר שיסמל עבורינו פונקציה:

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

כעת יש לנו תבנית חדשה בשפה שלנו.

A form for calling functions

כפי שראינו במקרה של ביטויי with, אנחנו גם צרכים תבנית חדשה על מנת שנוכל להשתמש בפונקציות שהגדרנו זה עתה. אנחנו נתשמש ב "call" לביצוע משימה זו. לדוגמה:

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

נשים לב שהביטוי הנ"ל למעשה יהיה זהה לביטוי with שהתחלנו איתו. ביטוי fun הוא למעשה כמו ביטוי with שאין לו ערך החזרה:

```
{with {x 5}
    {* x x}}
```

עד כה, לא ראינו הרבה תועלת בהוספת הפונקציה לשפה שלנו. מה שאנחנו באמת מפספסים זה הדרך לתת "שם" לפונקציות האלה. אם נקבל כללי הערכה נכונים נוכל להעריך ביטוי fun לערך כלשהו שיאפשר לנו לקשור אותו למשתנה באמצעות with. לדוגמה:

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

בביטוי הזה, אנחנו אומרים ש X הוא הפרמטר הפורמאלי או הארגומנט של הפונקציה, ו 5 ו 6 הם ה actual parameters שלנו.

מימוש הפונקציה

יש לנו תוכנית פשוטה, אך היא קשורה ישירות לאופן השימוש בפונקציות בשפה שלנו. אנו יודעים ש {call {fun {x} E1} E2} שווה לביטוי with, אבל מה שחדש כאן הוא שכאן אנחנו מאפשרים לכתוב רק את הביטוי fun בפני עצמו, ולכן אנחנו צרכים לתת לו משמעות כלשהי.

סמנטיקה של פונקציה

המשמעות או הערך של פונקציה בשפה שלנו צריכה להיו בערך כזאת: "ביטוי שצריך לחבר ערך עבור פרמטר X". במילים אחרות, בשפה שלנו יהיו סוגים חדשים של ערכים המכילים ביטוי שיהיה מוערך בהמשך, מעין הבטחה לחישוב.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

ישנן שלוש גישות בסיסיות המסווגות שפות תכנות ביחס לאופן ההתמודדות שלהן עם פונקציות:

1. first order: פונקציות הן לא ערכים אמיתיים. לא ניתן להשתמש בהן או להחזיר אותם על ידי פונקציות אחרות. המשמעות היא שלא ניתן לאחסן אותם במבני נתונים. כך היו רוב השפות "המקובלות" בעבר. קשה למצוא שפות כיום המשתמשות במודל זה.
2. High order: פונקציות שיכולות לקבל ולהחזיר פונקציות אחרות כערכים. כמו בשפת C.
3. first class: פונקציות הן ערכים שיש להן את כל ה"זכויות" שיש לערכים אחרים בשפה. בפרט, ניתן לספק אותם כקלט לפונקציות אחרות, להחזירן מפונקציות אחרות, לאחסן אותן במבני נתונים, וניתן לייצר פונקציות חדשות בזמן ריצה. לרוב השפות המודרניות משתמשות במודל זה.

The FLANG language

אנחנו נרחיב את השפה שלנו כדי לתמוך בפונקציות first class, בנוסף לכל הדברים שתמכנו בהם קודם לכן. ולכן אנחנו נשנה את השם של השפה שלנו מ WAE ל FLANG.
תחילה נעדכן את ה BNF שלנו:

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

בנוסף, נעדכן גם את הטיפוס שלנו ונוסיף את הבנאים המתאימים:

```
(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])
```

הפרסר לשפה החדשה:

```
(: parse-sexpr : Sexpr -> FLANG)
;; to convert s-expressions into FLANGs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
```

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

Subtitution rules for fun and call expressions

עלינו גם לתקן את פונקציית ההחלפה כדי שנוכל להתמודד עם התוספות.
הכללים הפורמאליים:

```
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}
```

והקוד המתאים לו:

```
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]
    [(Call l r) (Call (subst l from to) (subst r from to))]
    [(Fun bound-id bound-body)
     (if (eq? bound-id from)
       expr
       (Fun bound-id (subst bound-body from to)))]))
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

eval function for FLANG

שאלה: מה צריך להיות הערך המוחזר של הפונקציה eval ?
תחילה עלינו להחליט במה אנחנו משתמשים כדי לייצג ערכים בשפה שלנו.
לפני הוספת הפונקציות לשפה שלנו היו לנו רק ערכים מספריים והשתמשנו במספרים של racket על מנת לייצגם. כעת, לאחר שהוספנו ייצוג של פונקציות לשפה שלנו יש לנו שני סוגים של ערכים: מספרים ופונקציות.
נראה דיי קל להמשיך ולהשתמש בייצוג של racket עבור מספרים אבל מה נעשה עם הפונקציות ?
מה צריכה להיות התוצאה של הערכת הביטוי {{fun {x} {+ x 1}} ?
ובכן, זה אמור להיות הערך של הפונקציה, וזה משהו שניתן להשתמש בו כמו במספרים, אבל במקום פעולות חשבון אנחנו יכולים לקרוא לפונקציות האלה במקום. מה שאנחנו צרכים זו דרך להמנע מהערכת ביטוי גוף הפונקציה– "לעכב" אותה– ובמקום זאת להשתמש בערך כלשהו שיככיל את הביטוי המעוכב הזה באופן שיהיה ניתן להשתמש בו אחר כך.
הפתרון שלנו: eval יחזיר FLANG!
כדי להתאים לאפקט של העיכוב הרצוי, אנחנו דרכים בדרך כלשהי לשמור את המידע הבא אודות הפונקציה:

- ביטוי הגוף של הפונקציה שצריך להעריך אותו יותר מאוחר בזמן הקריאה לפונקציה.
- שם המזהה (formal parameters) שיש להחליף עם הקלט בזמן הקריאה לפונקציה.

נשים לב כי אובייקט התחביר המופשט שלנו (כלומר אובייקט מסוג FLANG) מכיל בדיוק את זה.
לפיכך, כדי לייצג פונקציה, אנו יכולים פשוט להשאיר את ה– FLANG המתאר אותה כפי שהיא.

כדי להיות בקנה אחד עם הטיפול בערכי המספרים, נשנה מעט את אסטרטגיית היישום שלנו: אנו נשתמש באובייקטים התחביריים שלנו למספרים ('(Num n) 'במקום רק' n ', שיהיה מעט לא נוח כשאנחנו נבצע את הפעולות האריתמטיות, אך זה יפשט את החיים על ידי כך שתאפשר להעריך פונקציות באופן דומה.

דוגמאות:

```
The above means that evaluating:

  (Add (Num 1) (Num 2))

now yields

  (Num 3)

and a number `(Num 5)' evaluates to `(Num 5)'.

In a similar way, `(Fun 'x (Num 2))' evaluates to `(Fun 'x (Num 2))'.
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

כללי הערכה הפורמאליים מטפלים בפונקציות כמו במספרים, ומשתמשים באובייקט FLANG כדי לייצג את שניהם:

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

הקוד המתאים עבור החוקים שלנו:

```
(: eval : FLANG -> FLANG)                        ; <- note return type
;; evaluates FLANG expressions by reducing them to *expressions*
(define (eval expr)
  (cases expr
    [(Num n) expr]                               ; <- change here
    [(Add l r) (arith-op + (eval l) (eval r))] ; <- change here
    [(Sub l r) (arith-op - (eval l) (eval r))] ; <- change here
    [(Mul l r) (arith-op * (eval l) (eval r))] ; <- change here
    [(Div l r) (arith-op / (eval l) (eval r))] ; <- change here
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (eval named-expr)))]           ; <- no `(Num ...)'
    [(Id name) (error 'eval "free identifier: ~s" name)]
    [(Fun bound-id bound-body) expr]             ; <- similar to `Num'
    [(Call (Fun bound-id bound-body) arg-expr) ; <- nested pattern
     (eval (subst bound-body                     ; <- just like `with'
                  bound-id
                  (eval arg-expr)))]
    [(Call something arg-expr)
     (error 'eval "`call' expects a function, got: ~s" something)]))
```

arith-op function

הפונקציה 'arith-op' תפקידה הוא לוודא שערכי הקלט הם מספרים (מיוצגים כמספרי FLANG), לתרגם אותם למספרים רגילים, ביצוע פעולת המתמטיות של racket, ולאחר מכן לעטוף מחדש את התוצאה ב- 'Num'.

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

פונקציית ה run

גם כאן עלינו להכניס מעט שינויים על מנת שנוכל להחזיר ערך מספרי:

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

יש עדיין בעיה עם הגרסה הנוכחית:

1. תחילה, אם call דומה לפעולת חשבות וגם ל with ומה שהוא עושה בפועל, למה אנחנו בכלל צרכים את התוספת הנ"ל.
2. שנית, מה **צריך** לקרות אם אנחנו מעריכים את הביטויים הבאים:

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

(run "{call {call {fun {x} {call x 1}}
                  {fun {x} {fun {y} {+ x y}}}}
            123}")
```

3. מה יקרה אם נעשה את הביטויים הנ"ל?

התיקון הפשוט הבא מתקן את הבעיה שנוצרה:

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

שפות תכנות – הרצאה 9

נזכר בחוקים שלנו לפונקציית 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})
                 = eval(Ef[eval(E2)/x])   if eval(E1) = {fun {x} Ef}
                           error!                 Otherwise =
```

כעת נראה את השלבים בפונקציית eval שהוגדרה קודם לכן באמצעות הדוגמה הבאה:
אנחנו נעקוב אחר הקריאות של הפונקציה eval

```
{ with { add2 { fun { x } { +  x { 1 1 }}}}
   { call add2 8 }}
```

לאחר שנפעיל את parse על הביטוי נקבל את הביטוי הבא:

```
(eval (with add2
  (Fun 'x (Add ('Id x)
    (Add (Num 1) (Num 1))))
  (Call (Id add2) (Num 8))) )
```

כאשר:

```
add2 – bound–id
Fun x (Add (Id x)
     (Add (Num 1) (Num 1))) – named–expr
Call (Id add2) (Num 8)) – bound– body
```

eval על named–expr, לשם נוחות נסמן את named–expr ב–T1:

```
T1 =  (Fun x (Add (Id x)
         (Add (Num 1) (Num 1))
(eval T1) → T1
```

הסבר: eval על Fun בעצם מחזיר לנו את אותו המבנה Fun. לכן eval על T1 יחזיר את אותו T1.
לשם נוחות, נסמם את ה–body ב–T2:

```
T2 = (Call (Id add2) (Num 8))
```

עכת, נצטרך לבצע החלפות, ישנה קריאה לפונקציה subst:

```
(subst (T2 add2 T1) ) → (Call T1 (Num 8))
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

כאשר:

add2– from
T1 — to

כעת, נעשה eval על הערך המוחזר הפעלת הפונקציה subst:

(eval Call T1 (Num 8))

כאשר:

T1– fun–expr
(Num 8)–arg–expr

נשים לב: שיש לנו עכשיו קריאה לפונקציה call ובתוכה יש קריאה ל eval שמקבלת את fun–expr.

fval = (eval T1)→ T1
(eval (Num 8)) → (Num 8)

כעת יש קריאה לפונקציה subst:

(subst (Add (Id ‘x)
(Add (Num 1) (Num 1))) ‘x (Num 8)) → (Add (Num 8)
(Add ((Num 1) (Num 1)))

כעת, נעשה eval על הערך המוחזר מהפונקציה subst:

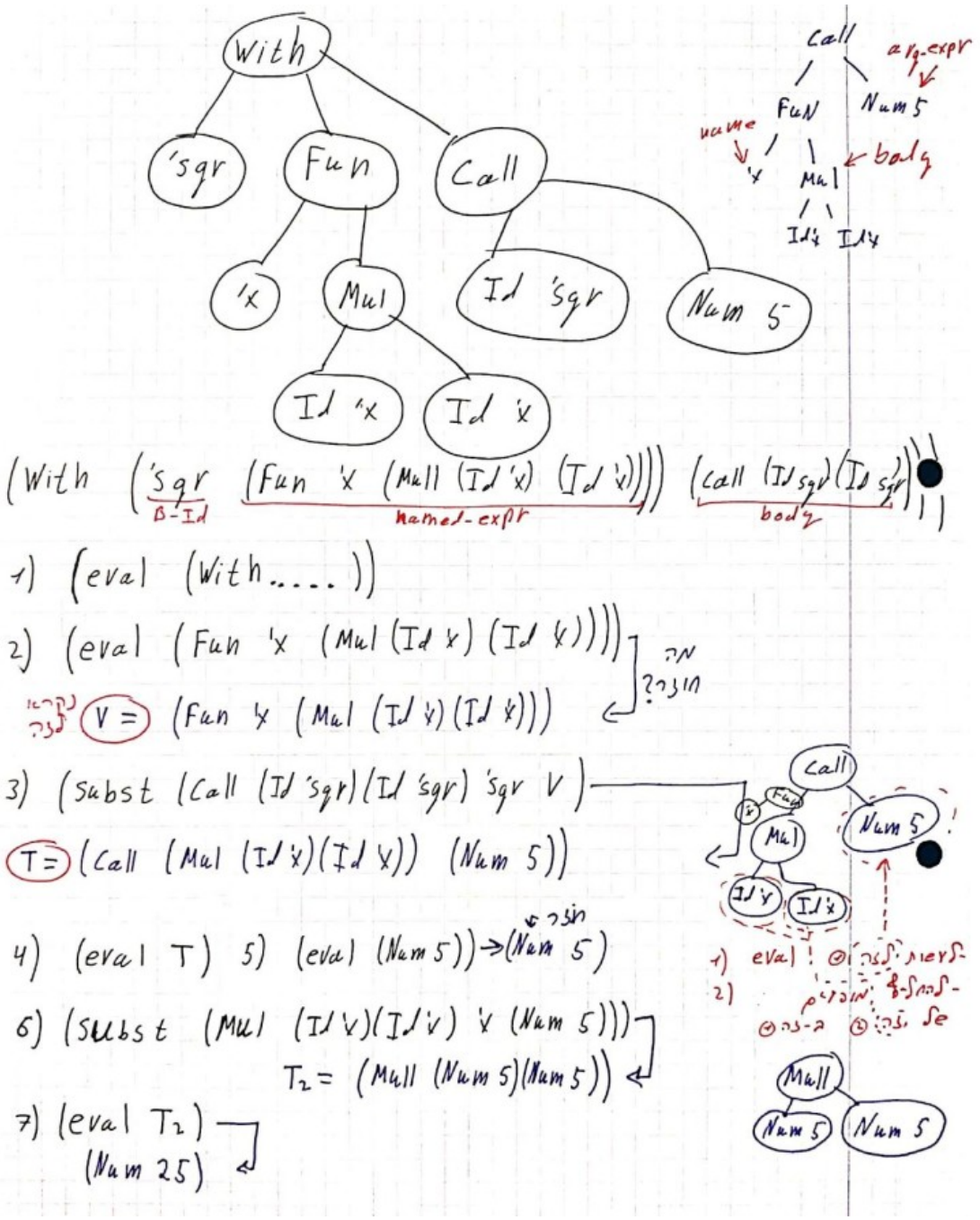
(eval (Add (Num 8((Add ((Num 1((Num 1(((→ ((Num 10(

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

דוגמה:



סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

המצב הנוכחי הוא לא יעיל. במצב הזה המתכנתים של השפה שלנו לא ירצו להשתמש בפונקציות שאנו מספקים. מדוע המצב הנוכחי לא יעיל? על פי הקוד שכתבנו נצטרך לעבור על הקוד כל פעם ולבדוק האם יש צורך לבצע החלפות.
נרצה לסרוק את הקוד פעם אחת בלבד. איך נעשה זאת?
נרצה להגדיר מבנה נתונים שנוכל להכניס אליו זוגות של שם וערך, שידמה פעולת מחסנית.

Initial Implementation of Cache functionality

תחילה, אנחנו זקוקים למבנה נתונים שיבצע את ההחלפה. לשם כך נשתמש ברשימה של רשימות של שני אלמנטים, השם והערך FLANG שלו:

```
;; a type for substitution caches:
 (define-type SubstCache = (Listof (List Symbol FLANG)))
```

אנחנו צרכים an empty subtitution cache, דרך להרחיב אותו, דרך לחפש את האלמנטים שבפנים:

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

למעשה, הסיבה להשתמש ברשימה של רשימות כזו היא של racket יש פונקציה מובנית בשם assq שתבצע חיפוש מסוג זה.
דוגמה:

```
> (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
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

חוקים פורמאליים עבור cached substitutions

צריך לשנות את חוקי ההחלפה כעת, ההערכה כוללת עתה "מטמון החלפה" שמתחיל ריק, ולכן פונקציית ההערכה צריכה לקבל ארגומנט נוסף, שהוא הזיכרון באותו רגע ולהרחיב אותו במידת הצורך.

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
```

נשים לב כי אין אזכור ל subst– וזאת מאחר שאנחנו באמת לא מחליפים אלא משתמשים במקום זאת ב cache. ה lookup וה extend מחליפים את כל המקומות שבעבר היה בהם שימוש ב subst.

Evaluating with substitution caches

קל מאוד ליישם את ה"הערכה" החדשה – היא מורחבת באותה דרך בה מורחב כלל ה"הערכה" הרשמי:

```
(: 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)
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

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

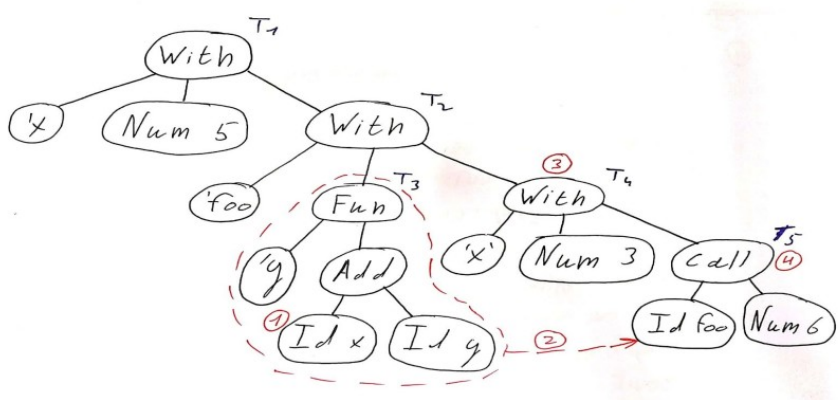
לבסוף, עלינו לוודא ש– "eval" נקרא בתחילה עם empty cache. קל לשנות את נקודת הכניסה העיקרית של פונקציות ה'run' שלנו:

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

נראה דוגמה עבור הביטוי הבא:

```
{ with { x 5}
        { with { foo { fun { y}} { +  x y}}}
                { with { x 3}
                        { call foo 6}}}}
```

אחרי שנעשה parse נקבל את העץ הבא:



סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

<u>נפרט בקצרה את השלבים להערכה של הביטוי על-פי מודל ההחלפות:</u>

1. Id X מוחלף ב- Num 5.
2. Fun לא משתנה ומחליפים את Id Foo ב- Fun.
3. ב- with אין מופעים חופשיים של x (ה-x מקושר ל- with הראשון) לכן הוא ישאר כמו שהוא כי לא תתבצע החלפה.
4. ה- call מקשר את Num 6 ל- y. (כי y הוא מופע חופשי ב- call). לכן, תתבצע החלפה של Id y ב- Num 6.

   ה- call מריץ את body ויוצא Num 11.

<u>הערכה של הביטוי על-פי מודל substitute cache:</u>

1. (eval T1 '())	SC0 = empty list
2. (eval (Num 5) SC0) → (Num 5)	
3. (eval T2 (extend 'x (Num 5) SC0))	SC1 = '(('x(Num 5)))
4. (eval T3 SC1) → T3	
5. (eval T4 (extend 'foo T3 SC1)	SC2 = '(('foo T3))
6. (eval (Num 3) SC2) → (Num 3)	
7. (eval T5 (extend 'x (Num 3) SC2))	SC3 = '(('x (Num 3))
8. (eval (Id foo SC3)) → T3	
9. (eval (Num 6) SC3) → (Num 6)	
10. (eval (Add (Id 'x) (Id 'y)) (extend 'y (Num 6) SC3))	SC4 ='(('y (Num 6))
11. (eval (Id 'x) SC4) → (Num 3)	
12. (eval (Id 'y) SC4) → (Num 6)	

Result → (Num 6+3 = 9) →

Dynamic and Lexical Scopes

זה נראה שזה היה אמור לעבוד, וזה אפילו עובד עבור כמה דוגמאות אבל לא עבור כולן. נראה שיש לנו באג..

עכשיו אנחנו מגיעים לבעיה טריקית שהיוותה בעיה במימושן של הרבה שפות. ננסה להריץ את הביטוי הבא ולהבין מה הוא יעריך:

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

אנחנו מצפים לקבל 7, אבל במודל החדש קיבלנו 9.

מה שהגענו אליו נקרא "dynamic scope". הסקופ נקבע על ידי הסביבה הדינמית בזמן ריצה (על ידי ה cache). זה כמעט תמיד לא רצוי. לפני שנמשיך, נגדיר את שני הסגנונות:

- dynamic scope: כל מזהה מקבל את ערכו מה scope שבו נעשה בו שימוש.
- Static Scope: כל מזהה מקבל את ערכו במקום שבו הוא הוגדר.

שפת racket היא סטטית, השפה שאחנו הגדרנו כעת היא דינאמית, השיטה הקודמת שמימשנו (החלפות) הייתה שיטה סטטית.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

ניתן להגדיר גם את pl להיות דינאמית:

```
#lang pl dynamic

  (define x 123)

  (define (getx) x)

  (define (bar1 x) (getx))
  (define (bar2 y) (getx))

  (test (getx) => ?)
  (test (let ([x 456]) (getx)) => ?)
  (test (getx) => ?)
  (test (bar1 999) => ?)
  (test (bar2 999) => ?)

  (define (foo x) (define (helper) (+ x 1)) helper)
  (test ((foo 0)) => ?)

  ;; and *much* worse:
  (define (add x y) (+ x y))
  (test (let ([+ *]) (add 6 7)) => ?)
```

סטטי מול דינאמי

- העובדה החשובה ביותר היא שאנחנו רוצים לראות את התוכנית שלנו מבעצת החלפות כמו ה subst הרגיל. המוטיבציה המקורית שלנו הייתה לייעל את ההערכה בלבד- ולא לשנות את הסמנטיקה. מכאן שאנחנו רוצים שהתוצאות של האופטימיזציה הזו תתנהג באותה הצורה. כל מה שאנחנו צריכים זה להעריך:

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

את הביטוי הנ"ל ל-7.

- זה לא מאפשר להשתמש בפונקציות כאובייקטים, למשל ראינו שיש לנו ייצוג פונקציונאלי לזוגות:

```
(define (kons x y)
  (lambda (n)
    (match n
      ['first  x]
      ['second y]
      [else (error ...)])))
(define my-pair (kons 1 2))
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

אם נעריך את זה בשפה דינאמית אנו מקבלים פונקציה כתוצאה, אבל הערכים המקושרים ל x ו y כברל א קיימים עוד. בעזרת מודל ההחלפות החלפנו את הערכים הללו, אך כעת הם הוחזרו ב cache ריק.

מימוש static scope: closures and enviroments

אז איך אנחנו מתקנים זאת ?

השורש של הבעיה: ה evaluator החדש לא מתנהג כקודמו.

במימוש הערכה הקודמת: פונקציות יכולו להתנהג כמו אובייקטים שזוכרים ערכים. לדוגמה:

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

התוצאה הייתה:

```
{fun {y} {+ 1 y}}
```

אם היינו קוראים לפונקציה כזו:

```
{with {f {with {x 1} {fun {y} {+ x y}}}}
   {with {x 2}
     {call f 3}}}
```

ברור מה תהיה התוצאה: f מחויב לפונקציה שמוסיפה 1 לקלט שלה, כך שהקריאה המאוחרת יותר ל- 'x' בכלל לא משפיעה.

עם ה caching evaluator הערך עבור:

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

הוא פשוט:

```
{fun {y} {+ x y}}
```

איך שום מקום ששמרנו בוא את הערך 1 !!!! זוהי שורש הבעיה שלנו.

הפתרון: אנחנו כבר יודעים שאנחנו צרכים אובייקט שמכיל את הגוף ואת רשימת הארגומנטים. מכיוון שאנחנו לא מבצעים חילופים- אנחנו בנוסף צרכים לזכור שאנחנו עדיין צרכים להחליף את X ב 1.
הפרטים שאותם עלינו לדעת הם:

- formal argument(s): {y}
- body: {+ x y}
- pending substitutions: [1/x]

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

נקרא לטיפוס המוחזר החדש שלנו closure מכיוון שהוא סוגר את גוף הפונקציה מבחינת ההחלפות שצריך לבצע.

השינוי הראשון: המידע שאנחנו רוצים להחזיק- closure צריכים את שלושת התנאים שראינו קודם לכן.
שנית: כאשר קוראים לפונקציה- כאשר סיימנו להעריך את הארגומנטים של call אבל לפני שאנו משתמשים בפונקציה יש לנו שני ערכים – אין יותר שימוש ב cache הנוכחי: סיימנו להתמודד עם כל ההחלפות שהיו נחוצות על הביטוי הנוכחי – אנו ממשיכים כעת להעריך את גוף הפונקציה, עם ההחלפות החדשות לארגומנטים הרשמיים והערכים בפועל שניתנו. אבל הגוף עצמו של הפונקציה נשאר כפי שהיה קודם לכן, לפני ביצוע ההחלפות.

הגדרה מחדש של החוקים הפורמאליים

```
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}, sc>
eval({call E1 E2},sc1)
         = eval(Ef,extend(x,eval(E2,sc1),sc2))
                           if eval(E1,sc1) = <{fun {x} Ef}, sc2>
         = error!          otherwise
```

סביבות

כעת ה substitution caches שלנו הוא יותר מ"רק זיכרון" כעת הוא למעשה מחזיק "environment" של ההחלפות בהן יש להחליף ביטויים.

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

האלגוריתם להערכת call הוא:

1. f := evaluate E1 in env1
2. if f is not a <{fun ...},...> closure then error!
3. a := evaluate E2 in env1
4. new_env := extend env_of(f) by mapping arg_of(f) to a
5. evaluate (and return) body_of(f) in new_env

שים לב כיצד כללי הסקופ המשתמעים מהגדרה זו תואמים את כללי הסקופ שנכללו על ידי הכללים המבוססים על החלפה.

The closure/environment interpreter

טיפוס חדש עבור ערכים (גם עבור פונקציות):
המשמעות של שינוי זה היא שכעת לא נוכל להשתמש באותו טיפוס עבור תחביר פונקציות וערכי פונקציות מאחר וערכי פונקציות כוללים יותר מסתם תחביר. יש לזה פתרון פשוט – לעולם איננו מבצעים החלפות, לכן איננו צריכים לתרגם ערכים לביטויים – אנו יכולים ליצור טיפוס חדש לערכים, נפרד מהטיפוס עצי התחביר המופשטים.
כשאנו עושים זאת, נתקן גם את הבאג שלנו בשימוש ב– FLANG בסוג הערכים: זו הייתה רק נוחות מכיוון שלטיפוס ה– AST היו מקרים לכל הערכים שהיינו צריכים. (למעשה, היינו צריכים לשים לב שגם Racket עושה זאת: מספרים, מחרוזות, בוליאנים וכו' משמשים בהם לתוכניות והן לייצוג תחביר – אך שימו לב שערכי הפונקציה אינם משמשים בתחביר.) כעת אנו מיישמים סוג 'VAL' נפרד לערכי זמן ריצה.

Implementing a new type for run–time values

טיפוס ENV:

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

הטיפוס VAL:

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

נתקן את הפונקציה lookup:

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

```
We don't need `extend' because we get `Extend' from the type definition, and we
also get `(EmptyEnv)' instead of `empty-subst'.
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

מימוש eval static scope:

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

אנחנו צרכים לעדכן את הפונקציה arith-op שתוכל לפעול על טיפוס מסוג VAL.

```
(: 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)))
```

כעת נשאר רק לעדכן את run, נראה כבר את כל הקוד:

```
---<<<FLANG-ENV>>>-----------------------------------------------------

  ;; The Flang interpreter, using environments

  #lang pl

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

```
            | { fun { <id> } <FLANG> }
            | { call <FLANG> <FLANG> }

Evaluation rules:
  eval(N,env)                = N
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({* E1 E2},env)        = eval(E1,env) * eval(E2,env)
  eval({/ E1 E2},env)        = eval(E1,env) / eval(E2,env)
  eval(x,env)                = lookup(x,env)
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)
           = eval(Ef,extend(x,eval(E2,env1),env2))
                             if eval(E1,env1) = <{fun {x} Ef}, env2>
           = error!          otherwise
|#

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

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

```
;; Types for environments, values, and a lookup function

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

(define-type VAL
  [NumV Number]
  [FunV Symbol FLANG 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 : (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))))

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

(: 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)])))
```

מה בעצם ראינו:

- ראינו שני מודלים סטטי ודינמי.
- ראינו שמודל ההחלפות הוא סטטי וה cache הוא דינאמי, וסיימנו במודל הסביבות שוהא מודל סטטי.

שפות תכנות – הרצאה 11

נבחן את הדוגמה הבאה במודל ההחלפות:

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

לאחר החלפה נקבל:

```
        {with {x 7}
          {call {fun {y} {+ x y}} 1}}}
```

לאחר החלפה נקבל:

```
    {call {fun {y} {+ 7 y}} 1}}}
```

ונקבל את התוצאה 8.

נריץ על פי מודל הסביבות:

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

נקבל שגיאה מכיוון שאין binding ל X כאשר הפונקציה מוגדרת.
כאשר הפוקציה מוגדרת לא מוגדל ערך כלשהו עבור המשתנה X ולכן יהיה שגיאה.
אבל במודל ההחלפות הקוד הזה עבד איך זה יתכן?
במודל ההחלפות היה לנו באג בקוד, שתוקן במודל הסביבות.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

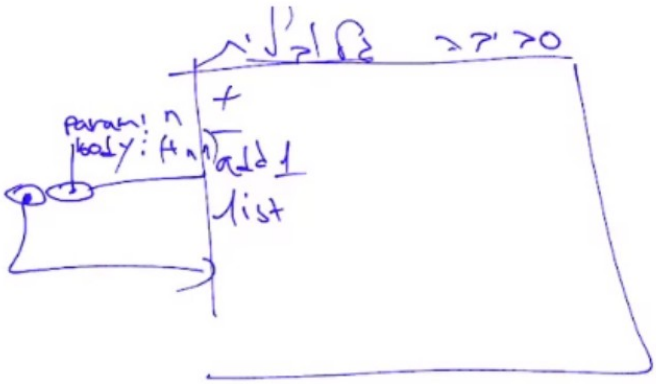
מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

שאלה אפשרית במבחן:

```
[(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))]
```

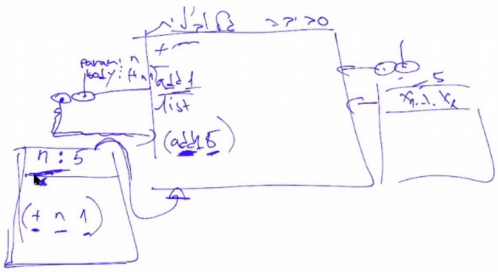
בקוד הנ"ל, על מנת להעביר אותו ממודל הסביבות למודל דינאמי, אילו שינויים בקוד יש לעשות ?
תשובה: כל שיש לעשות זה לשנות את f-env ל env, וזאת מאחר ש ש env לא תלוי ב scope.

איך זה עובד ב racket עצמה: יש לנו סביבה גלובאלית, שבה יש את כל הפונקציות:



קריאה ל add1, יוצרת אזור שמסתכל על שני דברים:

- מי ה param וה body
- מי הסביבה שלו- במקרה הזה הוא מצביע על הסביבה הגלובאלית, שהיא הסביבה שלו.



איך זה בעצם עובד: כאשר נקרא לפונקציה(add1 5) מה שיקרה בעצם זה שנוצרת הרחבה של הסביבה הגלובאלית, ההרחבה בודקת את מי היא מכירה, במקרה שלנו הוא יכיר את הפרמטר n שהוא 5, ואת המספר 1, שהוא מספר קבוע, ואז היא מגיעה לאופרטור +, והוא לא מכיר את האופרטור הזה, מה שהוא יעשה, הוא יחפש את ההגדרה לפונקציה הזאת בסביבה שלו שהיא כאמור הסביבה הגלובאלית, שם יתבצע תהליך דומה למה שאנו מתארים כרגע ולבסוף נקבל את התוצאה של 1 +5.

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

נניח ונרצה לבנות מנגנון שיעבוד באופן דומה ל cons שאנחנו מכירים מהרשימות, נקרא לו mycons והוא ייצור לנו זוגות:

- mycons יקבל שני מספרים, ויחזיר פונקציה שיודעת להחזיר מספר.

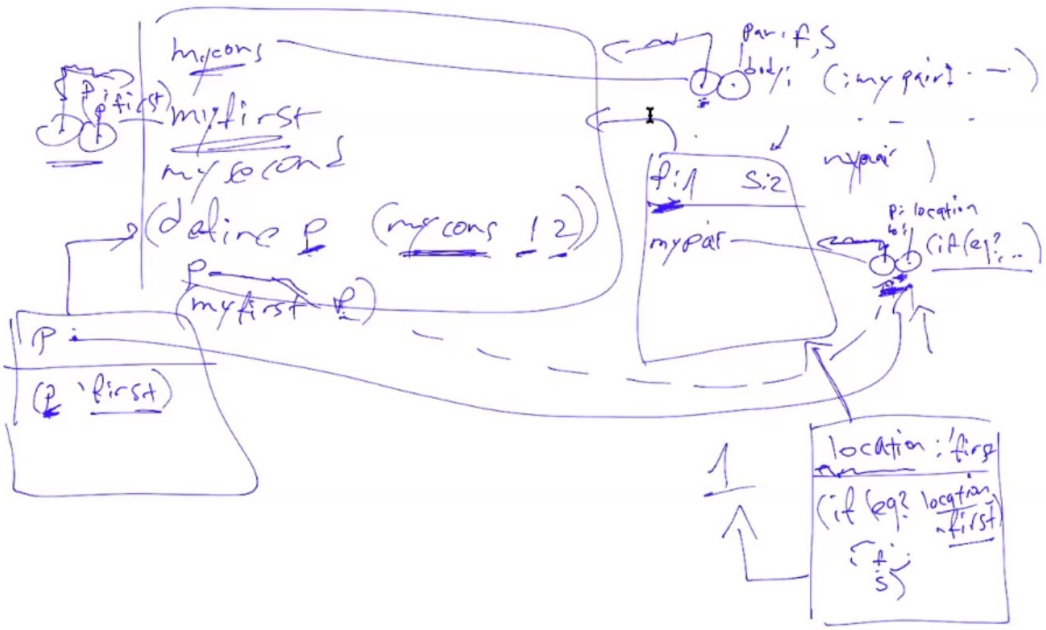
```
(: mycons : Number Number -> ((U 'first 'second) -> Number))
(define (mycons f s)
    (: mypair : (U 'first 'second) -> Number)
        (define (mypair location)
            (if (eq? location 'first) f s)) mypair)

;;
(: 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 בעצם ראינו? שעל ידי תכנות פונקציונלי אנחנו מקבלים אובייקט שאנחנו יכולים לעבוד איתו. נראה זאת בתמונה הבאה:



סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

במודל הדינאמי 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:

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

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

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 $PairV \to pair$:

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

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

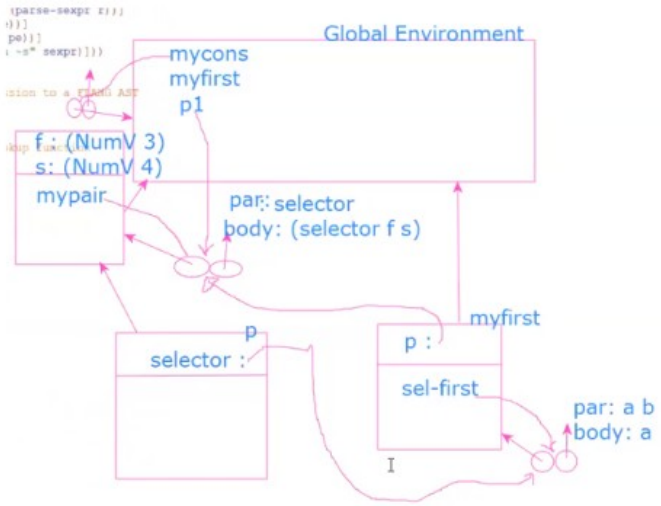
דוגמה:

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

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום



שפות תכנות – הרצאה 12

שאלה: מדוע ב eval בשורה המסומנת, אנחנו שולחים את arg–expr 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 שני 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}}}
```

סיכום קורס שפות תכנות

נכתב על ידי: שי נאור https://github.com/shaynaor

מבוסס על ההרצאות והמצגות של ד"ר ערן עמרי, גב' סואד תום

התשובה היא:

- שאנחנו ככה מרוויחים יעילות- כלומר על שליחת ה arg-expr אנחנו בעצם נשלח 10, ולא { 6 4 +} ששקול לשליחת עץ, יכול להיות מאוד יקר מבחינה חישובית.
- במודל הסביבות אנחנו מקבלים שהביטוי מסורש לפי הסביבה, אחרת היינו עלולים לבצע חישוב שגוי.

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

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

תשובה: נקבל פשוט 2 כי 1 הוא true ובכלל לא נגיע לחלק הקוד הבעייתי שדורש חלוקה ב 0.

אבל אם נגדיר:

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

אז הפעם נקבל שגיאה, כי כל העץ מורץ, גם c שהוא כאמור חלוקה באפס, היינו רוצים שגם אצלנו נוכל לממש lazy evaluation, כלומר שנעריך ביטויים רק שנצטרך לחשב. בשפת pl ישנה אומציה כזאת:

#lang pl lazy