שפות תכנות

מחברת מתוך ההרצאות של סעיד עסלי
אוניברסיטת אריאל, 2022

סיכום חלקי (חצי ראשון של הקורס)

שפות תכנות - מחברת הרצאות

נערך ונכתב על-ידי דור עזריה

סיכום חלקי (חצי ראשון של הקורס)

2022

ספר זה לא נבדק על ידי מרצה, יתכן שימצאו טעויות.

לסיכומים נוספים שלי במדעי המחשב ומתמטיקה:

https://dorazaria.github.io/

מוזמנים לעקוב אחרי 😊:

LinkedIn: https://www.linkedin.com/in/dor-azaria/

GitHub: https://github.com/DorAzaria

תוכן עניינים

הקדמה ל-Racket

יסודות של שפת תכנות

1. **תחביר** או *Syntax* - זו מערכת הכללים של השפה שקובעת מהו תחביר, מילה או ביטוי חוקי ומה לא.
2. **סמנטיקה** - כלומר משמעויות כמו למשל משמעות העומדת מאחורי מילה מסוימת בשפה.

ישנם 2 דרכים לכתוב תוכנה שמבינה סמנטיקה:

1. *Compiler*
2. *Interpreter*

הערה: *Compiler* מאוד קשה לכתוב.

הגדרות

- **קומפיילר** - או *מְהַדֵּר* (באנגלית: *Compiler*) הוא תוכנית מחשב המתרגמת משפת מחשב אחת לשפת מחשב אחרת. המהדר הקלאסי מקבל כקלט תוכנית הכתובה ב*שפה עילית* ומתרגם אותה לתוכנית ב*שפת מכונה*.
- **שפה עילית** - היא שפת תכנות המיועדת לשימוש על ידי מתכנתים אנושיים, בניגוד לשפות המוכוונות לשימוש על ידי כלים אוטומטיים (שפות ביניים) או שפות תכנות *low-level* המוכוונות להרצה על ידי מכונה (שפות אסמבלי ושפות מכונה).
- **שפת מכונה** - אוסף של הוראות המובן בצורה ישירה (ללא כל תרגום) על ידי המעבד של המחשב, ומבוצע על ידיו בעת פעולת המחשב. הוראות (פקודות) שפת המכונה הן רצף של סיביות. לסיבית שני מצבים אפשריים ועל כן ערכה מיוצג באמצעות הספרות אפס ואחת.
- **אינטרפרטר** - או מפרש (אנגלית: *Interpreter*) הוא תוכנה הקוראת תוכנית מחשב הכתובה בשפת תכנות ומבצעת אותה ישירות, פקודה אחר פקודה. שפה שהמימושים הנפוצים שלה הם בעזרת מפרש נקראת "שפה מפורשת" . דוגמאות נפוצות לשפות כאלה הן: *JavaScript*, פייתון, *bash* ועוד... הוא מקבל קוד בשפת תכנות כלשהי ומריץ אותה שורה אחר שורה. בסוף מחזיר את הפלט של הקוד. בשיטה שונה מה *compiler*; האחרון מתרגם את כל הקוד לשפה נמוכה יותר מהמקור, הראשון ממיר ומריץ כל שורה ושורה.

את הסמנטיקה של שפת תכנות נגדיר על ידי תרגום לשפה אחרת, כלומר על ידי כלי שמתרגם, כלומר אנחנו יוצרים משמעות לתוכן הכתוב - זה בעצם *הקומפיילר* ובקורס אנחנו נכתוב *אינטרפרטר* .
והכלי הזה נקרא ***Racket***.

הבסיס לשפת Racket

שפת *Racket* נוצרה בתור פלטפורמה לתכנון ומימוש שפות תכנות.
זוהי שפה פונקציונלית שמתייחסת לעולם כאבסטרקציה, ללא קשר לאיך הם ממומשים בזיכרון, והקוד "מסביר את עצמו" כי נדאג שלא יהיה *effect – side* ותמיד יהיה ערך מוחזר לעבוד איתו (ולא יהיה לנו דברים שקיימים מבחינתנו רק בזיכרון).
כל קובץ צריך להתחיל עם סוג השפה ובמקרה שלנו:

```
#lang pl
```

השפות מתחלקות לשני סוגים:

- ***אימפרטיביות*** – לא חייבות להחזיר ערך (יש פונ' *void*). השפה עוברת על הקוד שורה אחר שורה, ולכן כותבים קוד בצורת אלגוריתם. הכל מנוהל בזכרון בצורה מוגדרת היטב.
- ***פונקציונליות*** – הכל חייב להחזיר ערך, אין פונ' *void*. השפה לא עוברת על הקוד שורה אחר שורה, ולכן אי אפשר לכתוב אלגוריתם, אלא הכל נכתב באמצעות פונקציות. הזכרון לא מנוהל בצורה מוגדרת.

גרסת PL

PL – זוהי גרסה של *Racket* לסטודנטים.
כל מה שאנו כותבים בשפה, כל דבר ודבר הוא אובייקט בפני עצמו. 2 הוא אובייקט והוא שונה מ 3.
שניהם שייכים לאובייקט *integer* שהוא כולל את כל המספרים כולם.
String – כל סטרינג וסטרינג הוא אובייקט שונה אפ' אם הם מכילים אותו דבר.
Symbol – כל סימבול שיכיל את אותו הערך יהיה אותו אובייקט.

כתיבת ביטויים ואופרטורים

ב-*Racket* האופרטורים הם *Prefix* ועטופים בסוגריים, לדוגמה:

- חיבור: $(+\ \ 1\ \ 2)$
- חיסור: $(-\ \ 1\ \ 2)$
- שרשור מחרוזות: $(\ string-append\ \ "a"\ \ "b"\ )$

ביטויים ב-Racket

1. **עצמים** - נכתבים ללא סוגריים (כגון מספרים, מחרוזות וכו'...)
2. **ביטויים עם אופרטורים** - $(< expression_1 >\ \ < expression_2 > ...)$.

$< expression_1 >$ היא הפונקציה. לדוגמה: $+,\ \ *,\ string-append,\ \ >$.

$< expression_2 >$ הם הפרמטרים שנשלח לפונקציה (במקרה זה פרמטר אחד אך אפשר יותר).

3. **ביטויים מיוחדים המשתמשים במילים שמורות** -
   a. התניות - $(cond\ \ <exp>...)$ או $(if\ \ <exp>...)$
   b. השמה - $(define\ \ <name>\ \ <exp>)$ או $(let\ \ <name>\ \ <exp>)$
   c. הכרזה - $(:\ \ <name>\ :\ ...)$
   d. ועוד ...

כתיבת הערות

אפשר לכתוב הערה בשורה בודדת בעזרת ;; כאשר שום דבר אחר לא נמצא יחד עם ההערה הזאת.

```
;; comment
```

אפשר גם לכתוב הערה בהמשך לשורת קוד בעזרת ; לדוגמה:

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

בנוסף אפשר גם ככה:

```
#|
      This program does...
|#
```

תרגיל

נניח ש-2 הצלעות הקצרות של המשולש הם באורכים 3 ו-4. מה האורך של הצלע הארוכה במשולש?

נשתמש במשפט פיתגורס:

```
#lang pl
(sqrt (+ (* 3 3) (* 4 4) ) )       ; the answer is 5.
```

כלומר: $5 = \sqrt{3^2 + 4^2}$.

פירוש ביטויים

השפה *Racket* מפרשת ביטויים במעבר על שורת הקוד משמאל לימין.

בכל פעם שהוא רואה סוגריים הוא נכנס פנימה, מסיים את העבודה על הסוגריים, יוצא ממנו וממשיך ימינה.

<u>*דוגמה בשלבים*</u>:

$(+\ 2\ (*\ \ 3\ \ 4)\ (-\ \ (+\ 1\ \ 2)\ \ 3))$

$(+\ \ 2\ \ 12\ \ (-\ \ (+\ 1\ \ 2)\ \ 3))$

$(+\ \ 2\ \ 12\ \ (-\ \ 3\ \ 3))$

$(+\ \ 2\ \ 12\ \ 0)$

14

התניות

נכתוב בצורה הבאה:

```
(if
      <condition-expr>
      <positive-expr>
      <negative-expr>)
```

- הביטוי `<condition-expr>` הוא בעצם השאלה שלנו (חייבת להיות בוליאנית).
- הביטוי `<positive-expr>` הוא מה שנעשה במידה וקיבלנו *TRUE*.
- הביטוי `<negative-expr>` הוא מה שנעשה במידה וקיבלנו *FALSE*.

לדוגמה:

```
(if (< 2 3) 10 20)
```

אנחנו נשאל $2 < 3$, אם זה נכון אז נחזיר 10 ואחרת נחזיר 20.

התניה מרובה

```
(cond
      [<condition> <to do expression>]
      ...
      [<condition> <to do expression>]
      [else <else expression>])
```

בדומה ל-*else-if* ברגע שהגענו לבלוק [] שהביטוי שלו נכון אנחנו נעצור ולא נמשיך הלאה...

לדוגמה:

```
(cond
      [( eq? 'a 'b ) 0]
      [( eq? 'a 'c ) 1]
      [else 2])
```

נשאל האם $a == b$ וזה לא נכון, לכן נקפוץ לחלק הבא ונשאל האם $a == c$ וגם זה לא נכון ולכן נחזיר 2.

נקודה חשובה - ההזחות כאן כמו בפייתון הם בעלי חשיבות!, אי אפשר לכתוב את כל הביטוי בשורה אחת אלא אנחנו חייבים תמיד לרדת בכל שורה בדיוק כמו שכתוב כאן בדוגמאות.

בנוסף, גם לרווחים כאן יש חשיבות ואם לא נפריד עם הרווחים נקבל שגיאות.

דוגמה נוספת:

```
(cond
      [( and #t #f ) 1]
      [( or #t #f )) 2]
      [else 3])
```

- הביטוי ***and*** כמו באופן לוגי, כשנעבור משמאל לימין, מיד ברגע ונפגוש ב-*False* אז נצא מהבלוק.
- הביטוי ***or*** כמו באופן לוגי, כשנעבור משמאל לימין, מיד ברגע ונפגוש ב-*True* אז נעצור את *cond*.

יש לנו כאן אופטימיזציה כי אנחנו לא נעבור על כל השורה אם לא צריך, למשל ב-***and*** אם נפגשנו ב-*False* אז אין לנו צורך להמשיך לבדוק את המשך התנאי כי הוא לא רלוונטי וזה חיסכון.
במקרה של הדוגמה הזאת נחזיר את 2.

הגדרת קבועים

```
(define <name> <expression>)
```

- הביטוי `<name>` זה שם הקבוע
- הביטוי `<expression>` זה הערך

לדוגמה:

```
(define PI 3.14)
```

הגדרת פונקציה

```
(define (<function name> <arg1> ... ) <expression>)
```

- הביטוי `(<function name> <arg1> ...)` זה שם הפונקציה והארגומנטים
- הביטוי `<expression>` זה גוף הפונקציה

דוגמה:

```
(define (Not a)
      (cond
            [a  #f]
            [else #t]))
```

בדיקות

```
(: f : Number -> Number)
(define (f x) (+ x 1))

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

למשל בקטע קוד למעלה אנחנו משתמשים במילה השמורה ***test*** לצורך בדיקת הפונקציה *f*.
כאשר הביטוי *(f 0)* זה הדרך שבה אנחנו קוראים לפונקציה שקראנו לה *f* והכנסנו לה בארגומנט את *0*.
לאחר מכן היא מחזיר לנו ערך מסוים בטסט הזה אנחנו רוצים לוודא שהערך שחזר הוא שווה ל-*1*.
אם הוא היה מחזיר ערך שונה מ-1 אז הוא היה מתריע לנו על זה.

תרגיל

כתבו פונקציה של פיתגורס ואחרי זה כתבו טסט.

```
#lang pl

(: pyta : Natural Natural -> Number)
(define (pyta x y) (sqrt (+ (* x x) (* y y))))

(test (pyta 3 4) => 5)     ; sqrt(3^2 + 4^2) = 5
```

רשימות

רשימה זה אובייקט שמוגדר באופן רקורסיבי:

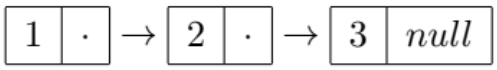
- רשימה ריקה נכתוב: null או ()' (עם הגרש)
- זוג/צמד (pair/cons) שהאיבר השני בו הוא רשימה.
- אם אנחנו רוצים לדעת שהגענו לסוף הרשימה אז נכתוב: null?
- רשימה זה הגדרה רקורסיבית של צמד (cons) אשר האיבר השני בו הוא רשימה או איחוד איברים בעזרת הפונקציה list.
- ה-cons זה צמד שמורכב מהאיבר הראשון והיתר שלו.

<u>*דוגמאות*</u>:

בדוגמה הבאה, יש צמד כאשר cons 1 זה איבר יחיד ו- (cons 2 null) הוא היתר.
שאר הדוגמאות הם אותו הדבר פשוט כתובות בצורות שונות:

- (cons 1 (cons 2 null))
- (cons 1 (cons 2 '()))
- (list 1 2)
- '(1 2)
- (cons 2 1); **not valid in PL!**

ניתן להסתכל על זה בצורה יותר מוכרת:



בדיקות על רשימות

כדי לבדוק האם אובייקט מסוים הוא מסוג רשימה ניתן להשתמש בפונקציה ?list.

דוגמאות:

- > (list? ‘(1 2))
- > (list? (cons 1 (cons 2 ‘()))))

זוג ב-PL

כדי לבדוק האם אובייקט מסוים הוא צמד ניתן להשתמש בפונקציה ?pair.

דוגמאות לבדיקה:

- > (pair? 1) // returns #f
- > (pair? (cons 1 (cons 2 ‘()))) // returns #t
- > (pair? (list 1 2 3)) // returns #t
- > (pair? ‘(1 2)) // returns #t
- > (pair? ‘()) // returns #f because that’s null
- > (pair? ‘(‘())) // returns #t because the first item is an empty list and the rest is null
- > (pair? ‘(1)) // returns #t

פונקציות נוספות לרשימה

ברשימה יש את האיבר הראשון והיתר, אם נרצה לעבור על הרשימה נרצה לעבור על הראשון והיתר וככה הלאה באופן רקורסיבי.

- פונקציית **first** - מחזיר את האיבר הראשון ברשימה.
- פונקציית **Rest** - מחזיר רשימה ללא האיבר הראשון. (מחזיר את היתר).

דוגמה:

(list 1 2 3 null)

אז ה-first יתן לי את 1 וה-rest יחזיר רשימה של $(2\ 3\ null)$.

תרגיל:

נרצה לקבל רשימה כ-input ונרצה להחזיר את האורך שלה.

בעצם נרצה לעבור על הרשימה וכל איבר שנראה נעשה לו 1+ וברגע שנגיע לרשימה ריקה, נחזיר 0 ואז נחזור אחורה כי list בנוי בצורה רקורסיבית.

פתרון:

Get length of a list

```
1:  #lang pl
2:  (: list-length : (Listof Any) -> Natural)
3:  (define (list-length list)
4:        (if (null? list)
5:              0
6:              (+ 1 (list-length (rest list)))))
7:  (test (list-length '(1 2 3 4)) => 4)
8:  (test (list-length '(1 2 'a 'b true)) => 5)
9:  (test (list-length null) => 0)
```

Get length of a list - הסבר

1:	
2:	הכרזה על הפונקציה, סוג הקלט וסוג הפלט. במקרה זה לא אכפת לנו איזה סוג איברים יש ברשימה ולכן הכנסנו כאן Any והערך שחוזר יהיה מספר טבעי.
3:	כל פעם שנהיה בתוך הפונקציה הקלט יהיה list
4:	נרצה לעבור על הרשימה וכל פעם אם נראה איבר נעשה 1+ ונמשיך ליתר הרשימה ואם הגענו לסוף הרשימה אז נחזיר 0 ונחזור ברקורסיה ולכן בכל שלב נבדוק, אם הגענו לסוף הרשימה אז:
5:	אז נחזיר 0
6:	אם לא, זה אומר שיש לי לפחות איבר אחד ברשימה לכן נעשה 1+ ובצורה רקורסיבית נמשיך עם מה שיש ביתר של הרשימה בעזר שימוש ב-rest
7:	ביצוע בדיקה, למשל אורך הרשימה הזאת הוא 4.
8:	ביצוע בדיקה, מספר האובייקטים ברשימה הזאת היא 5 וזה תקין כי הרי הגדרנו את הפונקציה להיות Listof Any לכן זה גם תקין.
9:	בדיקה על רשימה ריקה.

ההבדל בין רקורסיה לרקורסיית זנב

רקורסיה חוזרת עד לתנאי עצירה ולאחר מכן מבצעת את החישוב.
ברקורסיית זנב, הזנב מציין שנרצה להחזיר תשובה ישר בסוף התהליך, כלומר לשמור בכל שלב את התשובה שלי ונתקדם ונשמור בהתאם עד שנגיע לתנאי העצירה וישר נחזיר את התשובה.

דוגמה לחישוב עצרת ברקורסיה:

Factorial - Recursive

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

רקורסית זנב בדרך כלל דורשת פונקציית עזר, כי אנחנו נרצה בכל שלב להחזיק את התשובה ולעדכן בהתאם. מה שנרצה לעשות זה פונקצית עזר שהיא המעטפת והפונקציה הרקורסיבית תחזיק את המספר ואת הערך הרקורסיבי בכל שלב.

Factorial - Tail Recursive

```
#lang pl
(: 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))
```

Factorial - Tail Recursive - הסבר	
1:	
2:	
3:	הפונקציה helper מקבלת איבר n ואת הצובר acc שנצבר ומתעדכן בכל שלב.
4:	בכל שלב נשאל האם n הוא אפס,
5:	במידה והוא אפס אז נחזיר את ה-acc שהוא תמיד מעודכן בכל שלב
6:	אם לא הגענו לאפס אז נמשיך בשלבים, נקרא ל-helper כאשר נקטין את ה-n בצד אחד ונעדכן את הצובר להיות מה שהוא היה כפול ה-n הנוכחי,
7:	זה פונקצית המעטפת שלנו, היא מקבלת מספר טבעי ומחזירה טבעי.
8:	הפונקציה מתחילה לעבוד עם המספר n
9:	נקרא ל-helper עם n ועם 1 כי אם נתחיל ב-0 ונבצע הכפלות כל שאר החישובים יהיו גם 0.

<u>סימולציה בשלבים של העצרת ברקורסית זנב:</u>

- Step 1: (helper 3 1) → n != 0 then (3-1, 1*3)
- Step 2: (helper 2 3) → n != 0 then (2-1, 2*3)
- Step 3: (helper 1 6) → n != 0 then (1-1, 1*6)
- Step 4: (helper 0 6) → n == 0 then return acc=6

הפונקציה לא תחזור אחורה אלא היא תחזיר ישר את התשובה וכל השלבים אחורה נמחקו בזיכרון.
כלומר, יש כאן יעילות כי אין צורך לחזור אחורה וככה אין לנו בעיה של זיכרון.

Fibonacci Recursive

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

Fibonacci - Tail Recursive

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

פונקציות על רשימות

- פונקצית **append** מוסיפה ערכים לרשימה.

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

- פונקציות גישה על ידי **first, second, third** לשלושת המקומות הראשונים ברשימה בהתאם.
- פונקצית **rest** זה ההמשך של הרשימה.
- פונקצית **list-ref** זה גישה לפי אינדקס.

הגדרת אובייקטים

ניתן להגדיר אובייקטים בעזרת **define-type**, לדוגמה:

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

- ניתן להתייחס ל- Snake, Tiger כבנאים של האובייקט Animal שמקבלים ערכים ומחזירים אובייקט כזה מסוג Animal.
- מאחר ו-Animal הוא אובייקט אפשר לשאול עליו כמו למשל ?Animal שואלת האם האובייקט הוא מסוג Animal.

```
( Animal ? ( Snake ‘Slimey 10 ‘rats ) ) ; // returns #t
( Animal ? ( Tiger ‘Tony 12) ) ; // returns #t
( Animal ? 10) ; // returns #f
```

שימוש ב-Cases

זו בעצם צורה מיוחדת שמאפשרת לבדוק איך האובייקטים בנויים.

הוא מקבל אובייקט והוא ירצה להבין אם הוא מהצורה X או Y או כל צורה נוספת אחרת של האובייקט.

כלומר, ה-cases מאפשר לי:

1. לשאול מאיזה וריאנט האובייקט שקיבלתי
2. לייצר קישור בין הארגומנטים האקטואלים שאיתם נוצר האובייקט לשמות מזהים חדשים.

למשל עבור הדוגמה למעלה עם ה-Animal אפשר לבדוק איך בנינו אותו.

כלומר אנחנו נקבל Animal ובעזרת Cases נדע שהוא נוצר מהוואריאנט (בנאי) מסוג Snake.

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

כאשר ה-**n** הוא binding שהפונקציה מייצרת בין הערך לסוג האובייקט ולכן במקרה הזה:

n = 'Slimey , w = 10 , f= 'rats

- נשים לב שפה לא צריך else כמו שעשינו ב-cond בגלל שפה אנחנו יודעים בדיוק מי הבנאים שלו אז ניתן לוותר עליו, אבל יש מקרים שכן נוסיף.

דוגמה כוללת:

Cases Full Example

```
1:  #lang pl
2:  (: animal-name : Animal -> Symbol )
3:  ( define ( animal-name a )
4:        ( cases a
5:              [( Snake n w f ) n ]
6:              [( Tiger n sc ) n ] ) )
```

הסבר קוד: קראנו לפונקציה בשם animal-name שלוקחת Animal ומחזירה Symbol. כעת בשביל לדעת מהו האובייקט שקיבלנו נשתמש ב-cases.

שימוש ב-All

זה סינטקס מיוחד לטיפוסים. אנחנו יכולים לעשות Template כלומר פונקציה פולימורפית שיכולה לקבל כל טיפוס שנרצה בעזרת **All**.

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

כשנקרא לפונקציה אנחנו נצטרך לקרוא לה עם פקדירט שמקבל A ומחזיר Boolean וגם תקבל רשימה שכל האיברים בה הם מסוג A והיא תחזיר Boolean.
כלומר, הגדרנו סוג של Template כאשר אנחנו לא חייבים להחליט כרגע מה הוא A אבל ברגע שהחלטנו, הפונקציה every? תצפה לקבל את אותו האובייקט לשאר הפונקציות.

למשל, אם A הוא Natural אז הפונקציה every? מצפה לקבל פונקציות מסוג:

Natural → Boolean

Listof Natural → Boolean

All - Example

```
1:  #lang pl
2:  ( : every? : ( All ( A ) (A - > Boolean ) (Listof A) - > Boolean ) )
3:  ;; Returns true if all pass pred
4:  (define (every? pred lst)
5:        (or (null? lst)
6:              (and (pred (first lst))
7:                   (every? pred (rest lst))))))
```

All - Example - הסבר קוד

1:	
2:	
3:	נחזיר True אם כל האיברים ברשימה עוברים את הפרדיקט הזה (pred)
4:	הפונקציה מקבלת פרדיקט (pred) וגם רשימה (lst)
5:	<u>טיפול במקרה בסיס</u>: ה-or הוא צורה מיוחדת, אם קרה משהו שמחזיר ערך "1" כלומר ערך true כלשהו אז נעצור ונחזיר אותו. אחרת, נבדוק הלאה. לכן, במקרה שלנו, אם הרשימה ריקה אז נחזיר true.
6:	<u>*טיפול באיבר הראשון*</u>: אחרת, נדרוש 2 דברים, אנחנו נרצה גם שהפרדיקט (pred) מסכים על ה-first וגם (שורה 7):
7:	<u>*טיפול בהמשך הרשימה*</u>: וגם רקורסיבית עם אותו pred נפעיל את every? על המשך הרשימה.

שימוש ב-Let

ה-Let זה הגדרת שם מזהה לבלוק לוקאלי.

```
( let ([<id> <expr>] *) <expr>)
```

מה שזה בעצם אומר זה שבתוך הבלוק של ה-let אפשר להגדיר קבועים ואז נוכל להשתמש באותם הקבועים בתוך הגוף (body).

let - Example 1

```
1:  #lang pl
2:  (let ([x 10]
3:        [y 11])
4:        (+ x y))
```

let - Example 1 - הסבר

1:	
2:	נגיד כאן let כאשר הבלוק הראשון אומר x=10
3:	נגדיר בבלוק השני y=11
4:	החלק הזה שנקרא גם "גוף ה-let" יבצע את החישוב x+y=10+11=21

let - Example 2

```
1:  #lang pl
2:  (let ([x 0])
3:        (let ([x 10]
4:              [y (+ x 1)])
5:        (+ x y)))
```

let - Example 2 - הסבר

2:	מתחילים עם let ובבלוק שלו נגדיר x=0.
3:	הגוף של ה-let החיצוני הוא בעצם גם let, כאשר אצלו x בבלוק הראשון יש x=10
4:	ובבלוק השני של ה-let הפנימי יש y=x+1 נשים לב שהערכים של x,y צריכים להיות מוגדרים לפני שהם מוכרזים לכן בשורה הזאת זה לא יכול להיות x=10 אלא ה-x שבחוץ ולכן y=0+1=1
5:	הגוף של ה-let הפנימי הוא בעצם חישוב של x+y שזה בעצם 11=10+1

הסימון ***let**** נוצר כדי לאפשר לערכים להיות מוגדרים מיד אחרי שנכריז עליהם. זה בעצם let מקונן.

let* - Example
1: #lang pl
2: (let ([x 0])
3: (let* ([x 10]
4: [y (+ x 1)])
5: (+ x y)))

let* - Example - הסבר	
מתחילים עם let ובבלוק שלו נגדיר x=0.	2:
הגוף של ה-let החיצוני הוא בעצם * let, כאשר אצלו x בבלוק הראשון יש x=10	3:
ובבלוק השני של ה-*let יש y=x+1 נשים לב שהערך של x מוגדר כעת מיד אחרי שהוא מוכרזים לכן בשורה הזאת אנחנו נחשב את ה-x שנוצר בבלוק הקודם של *let כלומר y=x+1=10+1=11	4:
הגוף של ה-*let הוא בעצם חישוב של x+y שזה בעצם 21=11+10	5:

שימוש ב-Match

הוא מקבל ערך מסוים ומנסה להתאים אותו לתבנית (pattern) מסוימת.

כלומר הפונקציה match מחזירה את הערך (result-expr) לתבנית הראשונה שמתאימה.

הוא בעצם מנסה לעשות true, הוא קצת מרמה כי הוא מבצע את הקישוריות מהpattern אל ה-value,

אם **לדוגמה** יש משתנה x כלשהו אז ה-value יתאים את עצמו ל-x הזה ויחזיר את הערך שלו.

```
( match value [ptrn1 exp1] [ptrn2 exp2] ... )
```

דוגמה:

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

בקוד הזה הוא יחזיר לנו את הערך 6.

- ה-value במקרה הזה הוא רשימה של 1,2,3
- ה-ptrn הוא (list x y z)
- ה-exp הוא (+ x y z)

הפונקציה רואה שהתבנית הראשונה (list x y z) אכן מתאימה ל-value שכן ה-value הנבדק זה בעצם רשימה של מספרים 1,2,3 והיא מחזירה את מה שמופיעה ב-result-expr שזה חיבור שלושת האיברים.

דוגמה לשימוש של Match בתבנית של Symbol:

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

- הגדרנו let שבו ה-id הוא foo וה-expr הוא 'x (סימבול x).
- הגוף של ה-let הוא בעצם match כאשר foo הוא מקושר לערך 'x לכן כשנכנס לתבנית הראשונה אנחנו נבדוק האם הערך שיש לנו ב-foo מתאים לערך שיש לנו שזה 'x .
- מאחר והערך בבלוק הראשון שווה ל-foo אז ההתאמה הזאת מצליחה ויחזיר לנו "yes".
- אם בבלוק הראשון היה לנו במקום 'x את 'y אז זה לא היה מצליח כי foo מקושר ל-'x.

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

- לכן במקרה הזה היינו עוברים לתבנית הבאה (של ה-else), המילה else זאת מילה שאפשר לקשר לערך של foo (למילה else אין ייחודיות, היא תבנית שתמיד מצליחה ותקבל הכל).
- אנחנו נקשר את else לערך של foo כלומר else מקושר ל-'x וזה מצליח ולכן זה יחזיר לנו "no".

דוגמה לחישוב הסכום של האיברים ברשימת מספרים

Match - Example

```
1:  #lang pl
2:  (: sum : (Listof Number) -> Number)
3:  (define (sum lst)
4:        (match lst
5:              [`() 0]
6:              [(cons h t) (+h (sum t))]))
```

Match - Example - הסבר

2:	הכרזה של הפונקציה בשם sum, הקלט הוא רשימה של מספרים והפלט יהיה מספר כלשהו.
3:	נקרא לפונקציה sum עם הרשימה lst
4:	נשתמש ב-match כדי לבדוק איך הרשימה נראית, נשאל האם יש לנו איברים ברשימה שאפשר לסכום או שהיא ריקה.
5:	אם הרשימה היא ריקה אז התבנית הזאת תעבוד וזה אומר שסיימנו לעבור על הרשימה ונחזיר 0
6:	אם התבנית הראשונה לא עובדת אז נעבור לתבנית הזאת, היא תעבוד אם יש לנו לפחות איבר אחד ברשימה. אם יש איברים ברשימה, אז h מקושר לאיבר הראשון ברשימה ו-t הוא יתר הרשימה. ואז אפשר יהיה לעשות חיבור של ה-h ויתר הרשימה כלומר t ימשיך לצעד הרקורסיבי הבא עם sum וככה נסכום עד שנגיע לסוף הרשימה.

תבניות מיוחדות של Match

תבנית בשימוש שלוש נקודות (מחזוריות):

בתבנית של רשימה ניתן להשתמש ב:"..." המייצג חזרה (או לא) על התבנית הקודמת, לדוגמה:

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

ברגע שהוא רואה את ה-"..." הוא רוצה לקשר את המחזוריות ולקשר ל-x,y כלומר x מקושר לרשימה וגם y מקושר לרשימה כלשהי.

$$x \rightarrow (1\ 3\ 5\ 7)$$
$$y \rightarrow (2\ 4\ 6\ 8)$$

ולאחר מכן, בעזרת append נשרשר את x ו-y ונקבל:

$$(append\ x\ y) \rightarrow (append\ (1\ 3\ 5\ 7)\ (2\ 4\ 6\ 8)) \rightarrow (1\ 3\ 5\ 7\ 2\ 4\ 6\ 8)$$

תבנית המתאימה תמיד:

```
( match value
      [id result-expr])
```

התבנית הזאת תמיד תעבוד, ה-id יהיה *מקושר* ל-value.

תבנית המתאימה תמיד אבל ללא קישוריות:

```
( match value
      [_ result-expr])
```

הקו התחתון אומר "מה שאתה רוצה", אבל אין לנו משהו לבצע לו קישוריות כי אין לו id.

תבנית המתאימה תמיד אבל לפי טיפוס מסוים:

```
( match value
      [(number : n) result-expr])
```

אפשר לעשות קישוריות אבל רק אם יש לנו ערך שנרצה מטיפוס מסוים.

כלומר רק אם ה-value הוא מטיפוס של number אז יתבצע קישור ל-n.

```
( match value
      [(symbol : s) result-expr])
```

וכאן, רק אם ה-value הוא מטיפוס של symbol אז יתבצע קישור ל-s.

במקרה הזה, אם value היה מספר אז הקישוריות לא היה מתבצעת.

תבנית המתאימה ל-and:

כלומר תבנית התאמה לשני תבניות par1 וגם pat2.

```
( match value
      [(and pat1 pat2) result-expr])
```

אנחנו נרצה שהתבנית תהיה מתאימה לשניהם, אז אנחנו נרצה שה-value יהיה מתאים גם ל-pat1 וגם ל-pat2.
כלומר במידה וזה יצליח נקשר גם את pat1 וגם את pat2 ל-value.

תבנית של or: כל מה שמתאים לקישור עם value יקושר אליו.

דקדוקים חסרי הקשר

השפה הבסיסית AE

השפה ***AE*** או ***A****rithmetic* ***E****xpression* כלומר שפה שתעסוק בביטויים אריתמטיים.
דקדוק ***BNF*** הוא קיצור של ***B****ackus* ***N****oun* ***F****orm* שבעזרתו נגדיר את דקדוק השפה (מחליט מה הם המילים ששייכים לשפה ומה לא).

דוגמאות למילים השייכות לשפה *AE* למשל: $5,\ 1+4-2,\ 2+3,\ \ldots$
כלומר כל צורה של חיבור וחיסור שייכת לשפה.

הגדרת הדקדוק

$$< AE > \overset{\text{def}}{=} < num > \ | \ < AE > \ + \ < AE > \ | \ < AE > \ - \ < AE >$$

כלומר, השפה AE היא תו לא סופי (Non-Terminal) ותוגדר מ:

1. מספר כלשהו: ($< num >$).
2. חיבור (נקרא תו סופי): ($< AE > \ + \ < AE >$).
3. חיסור (נקרא גם תו סופי): ($< AE > \ - \ < AE >$).

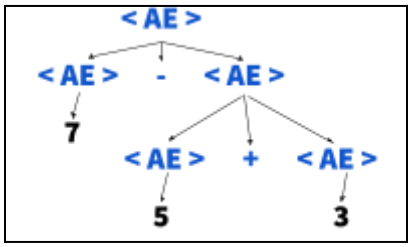
<u>דוגמה</u>

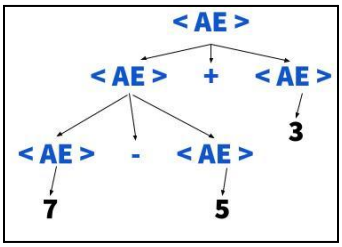
עבור הביטוי $1+4-2$ נגדיר:

$< AE >$
$\xrightarrow{3} < AE > \ - \ < AE >$
$\xrightarrow{2} < AE > \ + \ < AE > \ - \ < AE >$
$\xrightarrow{1} 1 + 4 - 2$

עץ גזירה

נסתכל על $7-5+3$ ונגזור את הביטוי בעזרת עץ גזירה לפי דקדוק השפה.





בעיה:

נשים לב שסדר הפעולות הוא מאוד שונה.

- בעץ השמאלי נקבל: $7 - (5 + 3) = -1$
- בעץ הימני נקבל: $(7 - 5) + 3 = 5$

לכן, חייבים להגדיר לדקדוק שלנו עבור איזה עץ אנחנו צריכים לפעול.

אם קיימת מילה שיש עבורה שני עצי גזירה שונים אז אנחנו צריכים להגדיר Ambiguity (דו-משמעות).

פתרון:

כשאנחנו מתכננים שפה אנחנו נעדיף להימנע מדו-משמעות.

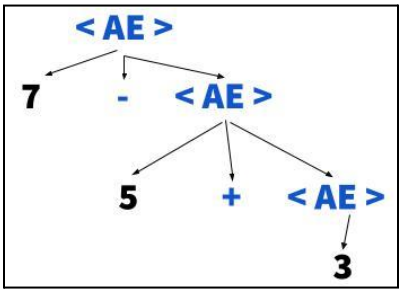
נפתור את הבעיה הזו על ידי שינוי ההגדרה של הדקדוק שלנו:

$$< AE > \stackrel{\text{def}}{=} < num > \ | \ < num > \ + \ < AE > \ | \ < num > \ - \ < AE >$$

כלומר, השפה AE היא תו לא סופי (Non-Terminal) ותוגדר מ:

1. מספר כלשהו: ($< num >$).
2. חיבור (נקרא תו סופי): ($< num > \ + \ < AE >$).
3. חיסור (נקרא גם תו סופי): ($< num > \ - \ < AE >$).

לא משנה איך נפרק את הדקדוק אנחנו תמיד נקבל את אותו עץ הגזירה וכעת לפי עץ גזירה אנחנו נקבל:



טיפול בקידומות

נניח שנרצה להתייחס לקידומות כלומר נרצה להגדיר את סדר פעולות החשבון:

$$< AE > \stackrel{\text{def}}{=} < num > \ | \ < AE > \ + \ < AE > \ | \ < AE > \ * \ < AE >$$

כדי לטפל בבעיית דו-משמעות נשנה לצורה הבאה:

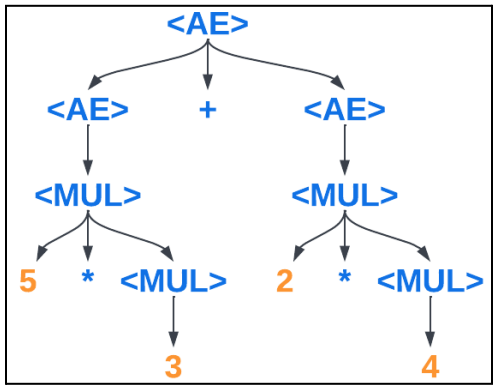
$$< AE > \stackrel{\text{def}}{=} < AE > \ + \ < AE > \ | \ < MUL >$$

נגדיר את כלל הכפל:

$$< MUL > \stackrel{\text{def}}{=} < num > \ | \ < num > \ * \ < MUL >$$

דוגמה:

נסתכל על $5 \cdot 3 + 2 \cdot 4$ ונגזור את הביטוי בעזרת עץ גזירה לפי דקדוק השפה.



פתרון בעזרת סוגריים

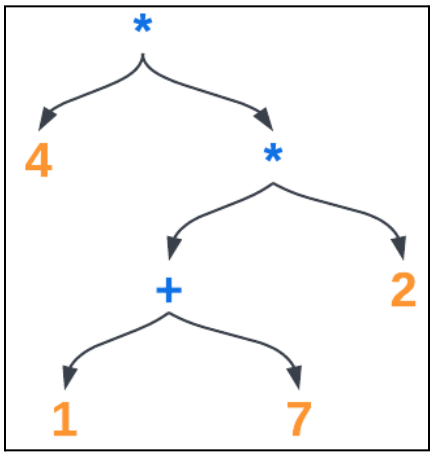
ראינו את הבעיות עבור דו משמעות וגם עבור קידומות.

כדי לפתור את בעיית דו המשמעות וגם את בעיית הקידומות נעזר בסוגריים.

דוגמה:

נסתכל בביטוי $4 \cdot ((1 + 7) \cdot 2)$ ונשאל איך נבנה לו את עץ הגזירה?

פעולות כפל יופעלו לפני החיבור ונקבל את העץ:



הגדרת הדקדוק עם סוגריים

$$< AE > \stackrel{\text{def}}{=} < num > \mid \{+ \quad < AE > \quad < AE > \} \mid \{- \quad < AE > \quad < AE > \}$$

לדוגמה: $\{+ \quad \{- \quad 7 \quad 5\} \quad 4\}$.

פרשן - Parser

פרשן

עד עכשיו התעסקנו רק בתחום התיאורטי, כתבנו תיאורטית את הדקדוק לשפה.

כעת נרצה לקודד את השפה ב-Racket בעזרת Parser.

- ה-Parser הוא בעצם "פרשן" שיש לו קלט ופלט.
- הפרשן מקבל String ומחזיר לנו עץ גזירה אבסטרקטי (Abstract Syntax Tree - ***AST***).
- אם קיבלנו String שלא שייך לשפה אנחנו נזרוק עבורו Exception.

שלבים:

1. *שלב הקריאה* - אני מקבל String, בודק אם ה-String חוקי ובמידה וכן הוא יתרגם ויחזיר מבנה נתונים מתאים לביטוי.
2. *שלב האקטואלי* - נמיר את מה שאנחנו קיבלנו מה-String וניצור ממנו עץ אבסטרקטי.

נקודה חשובה: שימוש באריתמטיקה כפרפיקס פותר לנו את בעית הדו משמעות.

כלומר במקום $< AE > \quad + \quad < AE >$ נכתוב: $\{+ \quad < AE > \quad < AE >\}$.

מעבר לעולם הפרקטי

קטעי הקוד בחלק זה הם רציפים - כלומר נציג כאן קוד שלם שמחולק לקטעים לטובת הסבר.

נרצה להגדיר את מבנה העץ *AST*, לכן תחילה נגדיר את האובייקט *AE* והבנאים שלנו *Num, Add, Sub*.

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

הגדרת Sexpr

- כל Number או Symbol הוא sexpr.
- כל רשימה עם איברים של sexpr הוא sexpr בפני עצמו.
- נכתוב אותו בצורה $string \; -> \; sexpr$.
- השיטה הזאת ממירה לנו את המחרוזת לצורה שאפשר לעבוד איתה ב-racket. כלומר, הוא יחזיר לנו מחרוזת שה-parser שלנו ידע לעבוד איתה וכך יהיה אפשר לבנות את עץ ה-AST.
- הוא לא רק ממיר לנו, הוא גם יחזיר לנו מחרוזות שגויות כמו למשל { {4 3 +} כלומר הסוגריים שגויות.

כעת נבנה את ה-Parser שמקבל Sexpr ומחזיר לנו AE כלומר "עץ".

לצורך ההסבר, נשתמש בדוגמה עבור מחרוזת מסוג: { {} 6 5 3 +} כלומר סכימה של 3 איברים.

לכן, ברשימה הזו חייבים להיות עד 3 איברים בלבד, אחרת ה-Parser לא ידע איך לעבוד עם הצורה הזאת.

```
(: parse-sexpr : Sexpr -> AE)
(define (parse-sexpr sxp)
  (cond
        [(number? sxp) (Num sxp)]

        [(and
          (list? sxp)
          (= (length sxp) 3)
          (eq? (first sxp) '+)
          )
         (Add
          (parse-sexpr (second sxp))
          (parse-sexpr (third sxp))
         )]

        [(and
          (list? sxp)
          (= (length sxp) 3)
          (eq? (first sxp) '-)
          )
         (Sub
          (parse-sexpr (second sxp))
          (parse-sexpr (third sxp))
         )]

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

הסבר:

1. אנחנו מגדירים parse-sexpr שמקבל Sexpr ומחזיר לנו את האובייקט שיצרנו AE.
2. נגדיר Parser עבור כל סוג בנאי שהצהרנו ב-define-type AE:
   a. עבור Num, אם ה-Sexpr הוא מטיפוס של Number אז פשוט נחזיר אותו כ-**Num**.
   b. עבור Add, אם sxpr הוא רשימה **וגם** אורך הרשימה הוא 3 **וגם** הפרפיקס של המחרוזת הוא פעולת החיבור "+" אז נשלח רקורסיבית את האיבר השני של הרשימה ובנפרד, נשלח גם את האיבר השלישי של הרשימה, וכעת בצעד הבא, האיברים האלו יהיו מספרים מסוג **Num** לכן הם יחזרו בצורה של **AE** ל-**Add** שהגדרנו אותו ב-define-type לקבל AE AE.
   c. עבור Sub, באופן דומה ל-Add רק שהפעם עם "-" וגם נשתמש ב-**Sub**.
3. ייתכן שהמחרוזת ששלחנו ל-Sexpr היא שגויה בסינטקס שלה (כמו סוגר פותח בלי סוגר), לכן אנחנו צריכים לדאוג למקרה של שגיאה ב-**else** עם ה-error. השימוש ב-s~ הוא כדי להציב בו את sxp.

נקודה חשובה - תפקיד ה-Parse הוא לא לחשב את מה שהוא תרגם, אלא רק לתרגם לצורה של השפה שאנחנו בונים. לדוגמה, אם נשלח ל-Parse את "4" הוא צריך להחזיר לנו (Num 4).

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

#| TESTS |#
(test (parse "4") => (Num 4))
(test (parse "{+ 3 4}") => (Add (Num 3) (Num 4)))
(test (parse "{+ 3 {- 5 4}}") => (Add (Num 3)
                                      (Sub (Num 5)
                                           (Num 4))))
(test (parse "{+ 2 3 4 5}") =error> "bad syntax")
```

הסבר: הגדרנו פונקציית parse שתקבל מחרוזת ותשלח אותה בצורה של sexpr לפונקציה parse-sexpr.
לאחר מכן, בוצעו בדיקות (טסטים) כולל מקרה של error, השימוש ב- =error> הוא כדי לבדוק אם חזר אלינו שגיאה בצורה של מחרוזת ומוכל בה הטקסט "bad syntax" אז זה סימן שהחזרנו נכון את מה שרצינו לבדוק.

אפשר לעשות את הקוד קריא יותר ונכון יותר, נשנה את parse-sexpr שכתבנו.
אנחנו נשנה לשימוש ב-match שפועל עבור כל מקרה (Num, Add, Sub) לפי התאמה לתבניות מוגדרות.

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

ה-else זה משתנה שהגדרנו שיכיל את שאר המשתנים שהם אינם מהצורה של התבניות שכתבנו.
חשוב לציין כי-else זה לא מילה שמורה, כלומר היינו יכולים לכתוב במקומו גם pikachu וזה היה עובד.

מעריך - Evaluator

ה-Evaluator מקבל צורה מוגדרת מה-Parser ומחזיר את ערך התוצאה.
ה-eval היא פונקציה שעושה "הערכה" לביטוי (לקוד).

נחזור להגדרת הדקדוק הראשונה:

$$< AE > \stackrel{\text{def}}{=} < num > \ | \ < AE > \ + \ < AE > \ | \ < AE > \ - \ < AE >$$

כלומר, השפה AE היא תו לא סופי (Non-Terminal) ותוגדר מ:

❶ הפעלת eval על מספר: $eval(< num >) = \ < num >$
❷ הפעלת eval על חיבור: $eval(E1 \ + \ E2) = eval(E1) \ + \ eval(E2)$
❸ הפעלת eval על חיסור: $eval(E1 \ - \ E2) = eval(E1) \ - \ eval(E2)$

נזכר בבעיית דו המשמעות, מה יקרה כאשר נחשב את $eval(7 \ - \ 4 \ + \ 2)$?
נשים לב שאפשר ליצור עבורו 2 עצי גזירה שונים עם תוצאות שונות.
וראינו בהרצאה קודמת שהפתרון עבורו הוא בעזרת שינוי הגדרה:

$$< AE > \stackrel{\text{def}}{=} < num > \ | \ \{+ \ < AE > \ < AE> \} \ | \ \{- \ < AE > \ < AE> \}$$

קומפוזיציונליות

נתבונן בביטוי הבא: $eval(\{+ \ E1 \ E2\}) \ = \ eval(E1) \ + \ eval(E2)$
נחלק ל-2 תרחישים:

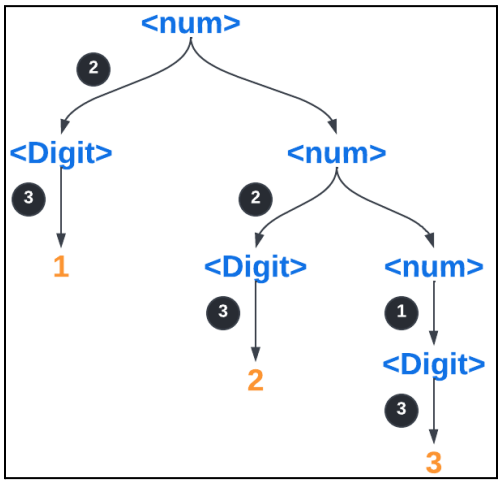
1. $E1 \ = \{+ \ \{- \ 7 \ 4\} \ 2\}$
2. $E1 \ = \ 5$

ב-eval לא אכפת לנו המבניות של העץ עבור $eval(E1)$ למשל, מה שאכפת לו זה הערך הסופי.
לכן, 2 התרחישים שהצגנו אומרים בשבילו אותו הדבר כי שניהם עם אותה תוצאה סופית.

הגדרה

$$< num > \stackrel{\text{def}}{=} < Digit > \ | \ < Digit > \ < num >$$
$$< Digit > \stackrel{\text{def}}{=} 1|2|3|4|5|6|7|8|9|0$$

דוגמה: נבנה עץ עבור המספר 123:

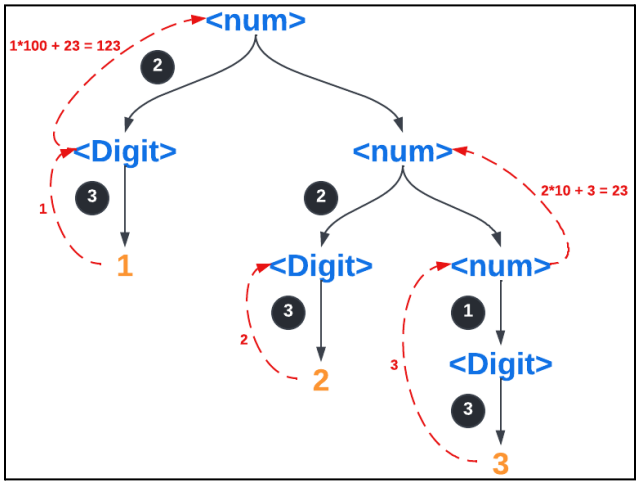


כעת לאחר שקיבלנו את העץ, נרצה לעשות eval לשפה הזאת.

נגדיר:

$$eval(< Digit >) \;=\; < Digit >$$

אך איך נחשב את הכפל?, נציג את החישוב החוזר מהעץ:



כלומר, כדי לחשב את 123, אנחנו צריכים לכפול כל גורם שחוזר ב-10 לפי גובה העץ.

במקרה זה, $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 = 123$.

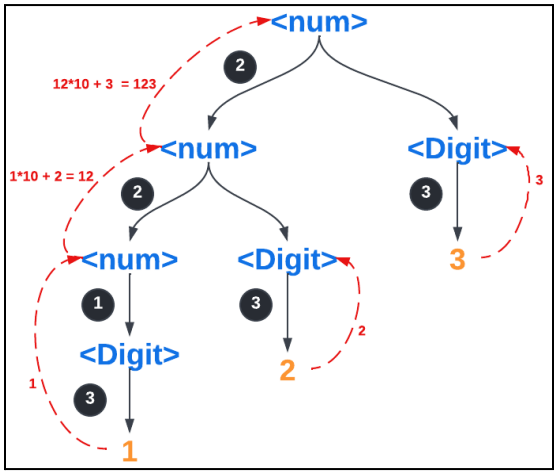
לכן, ה-eval עבור כפל מוגדר בצורה הבאה:

$$eval(< Digit >< num >) = \left(eval(< Digit >) \cdot 10^h\right) + eval(< num >)$$

בעיה: נשים לב שכעת יש לנו התחשבות בצורת העץ בגלל הגובה שלו, ולכן הוא לא מקיים את הקומפוזיציונליות. אפשר לפתור את הבעיה הזאת באמצעות החלפה של Digit ו-num מההגדרה הקודמת:

$$< num > \stackrel{\text{def}}{=} < Digit > \mid < num > \quad < Digit >$$
$$< Digit > \stackrel{\text{def}}{=} 1|2|3|4|5|6|7|8|9|0$$

כעת, בהמשך לדוגמה עבור 123, נקבל את העץ הבא:



נפתרה לנו בעיית הקומפוזיציונליות כיוון שבכל פעם שנחזור נכפיל ב-10 כמספר קבוע.
כלומר, אנחנו לא תלויים בעץ כהגדרת הקומפוזיציונליות ולכן, ההגדרה החדשה שלנו תהיה:

$$eval(< num >< Digit >) = (eval(< num >) \cdot 10) + eval(< Digit >)$$

קוד מודולרי

למה צריך את ההפרדה בין parsing לבין eval:

1. אם צריך להחליף בשלב מסוים אחרי שכתבנו את השפה ....
2. אנחנו רוצים לשפר את זמן הריצה של ה-eval אז נרצה לשנות את ה-eval ולא את ה-parser.

בקוד שבו אנחנו עושים את ההפרדה הזו נקרא **קוד מודולרי**.
כלומר, כל חלק עובד בנפרד eval, parsing.

בעולם הפרקטי

דיברנו על קוד מודולרי המפריד בין ה-Parser לבין ה-Eval.

ב-Racket, כדי להפריד ביניהם אנחנו נמנע מלכתוב eval יחד עם ה-Parser.

כלומר, אנחנו נחלק אותם ל-2 קטעי קוד שונים וכך נהפוך את הקוד שלנו למודולרי.

בפרק הקודם ב-Parser עבדנו על קוד ובנינו את הפונקציה parser.

כעת, נמשיך מאותה הנקודה שעצרנו עם אותו הקוד, ונצרף לו כעת את ה-Eval עבור השפה שלנו.

```
(: 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))]
   )
)

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

הסבר: אנחנו משתמשים ב-cases עבור התאמה של סוג בנאי, למשל במידה -expr הוא מהצורה (Num n) אז נחזיר את n.

ה-eval הוא הגורם המחשב (מעריך) שמקבל את עץ הגזירה (AST -> AE) ומחזיר לנו מספר Number.

חשוב לציין כי eval הוא לא שם שמור בשפה אלא זה מוסכמה בקורס עבור שם של Evaluator.

כעת, ניצור פונקצית מעטפת שתפעל בצורה נכונה יותר עבור כל תהליך השלם שלמדנו עד כה:

$$string \rightarrow parser \rightarrow eval \rightarrow Number$$

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

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

הוספת הפעולה - with

ניקח ביטוי מסוים:

{+ {* 3 4} {* 3 4}}

מה חסר לנו בשפה? מה היה לנו יותר טוב בשפה כדי לשפר את הביטוי הזה?
אם היה לנו את האופציה: $x = \{* \ 3 \ 4\}$ ואז $\{+ \ x \ x\}$ אז היינו יכולים למנוע חזרות.

נרצה להתחיל לאפשר לשפה שלנו את האופציה הזו, כלומר להוסיף מזהים לשפה שלנו.
אנחנו נציג אותה בצורה שונה מהדוגמה שהצגנו כאן, כלומר בהתאם ל-Racket:

{with {x {* 3 4}} {+ x x}}

ראינו בתרגול שימוש ב-let שמתעסק בקישור לוקאלי.
אנחנו נממש את with בעזרת קישור לוקאלי בדומה לנלמד עם let.

למה כדאי לנו להוסיף שמות מזהים?

1. *למנוע כפילויות של קוד* - במקום לשנות את אותו הערך במספר מקומות אפשר לשנות במקום אחד.
2. *יעילות* - מאפשר קוד קצר, לא צריך לחזור על חישובים מספר פעמים, אלא אפשר לחשב פעם אחת.
3. *פשטות וקריאות* - קוד פשוט וקריא מונע באגים וקל להבנת המתכנת.
4. *היכולת של המתכנת לבטא את עצמו* - יכולת למתכנת לתת ביטוי לשמות משתנים, למשל data = 4.

צעדי ביצוע

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

1. הוספה: חישוב 4+2
2. הצבה: הצבת החישוב ל-x כלומר x=6
3. הכפלה: חישוב x*x
4. הצבה פנימית: y = 36
5. הוספה: y+y
6. החזרה של החישוב.

למה with ולא let?

מאחר ואנחנו בונים שפה חדשה, אז ה-let לא מוכר בשפה שלנו אלא מוכר רק ב-Racket.
לכן, אנחנו יוצרים סוג של let משלנו, שנקרא לו with באותו השיטה כמו שעבדנו עד עכשיו עבור פעולות חשבון שיצרנו עד כה.

הוספת with לדקדוק של השפה

תחילה, לשפה שיצרנו עד כה AE נשנה את שמה ל- WAE על שם הוספת $with$.

$$\{with\ \{< id >\ \ < WAE >\}\ < WAE >\}$$

- כאשר $< id >$ מתייחס לכל Symbol של Racket, וגם $< num >$ זה כל Number של Racket.
- אנחנו מוסיפים לדקדוק פעולה נוספת בשם $< id >$ שנועד עבור הצבת ערך החישוב.

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

הוספה ל-parser:

```
(: parse-sexpr : Sexpr -> WAE)
(define (parse-sexpr sxp)
  (match sxp
    [(number: n) (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sxp
         [(list 'with (list (symbol: name) named-expr) body)
          (With name (parse-sexpr named-expr) (parse-sexpr body))]
          [else (error 'parse-sexpr "bad with syntax in ~s" sxp)])]
    [(list '+ l r) (Add (parse-sexpr l)(parse-sexpr r))]
    [(list '- l r) (Sub (parse-sexpr l)(parse-sexpr r))]
    [else (error 'parse-sexpr "bad syntax in ~s" sxp)]
  )
)
```

כתיבת ה-Eval עבור with

למשל עבור: $eval(\{with\ \{x\ E1\}\ E2\})$, מכמה שלבים מורכב ה-eval?

1. ***הערכה*** - כלומר $v \leftarrow eval(E1)$
2. ***הצבה*** - $E2' \leftarrow subst(E2,\ x,\ v)$
3. ***הערכה*** - $eval(E2')$.

נתחיל לממש תחילה את השלב השני (שלב ההצבה).

<u>הגדרה פורמלית (הגדרה לא תקינה)</u>:

נסמן ב- $e[v/i]$ שתפקידו: מחליף (substitute) את כל המופעים של i בתוך E עם הביטוי v.

<u>דוגמה 1</u>: נניח שיש לנו את : $\{with\ \{x\ 6\}\ \{*\ \ x\ x\}\}$, במקרה הזה, $i = x,\ e = \{*\ \ x\ x\},\ v = 6$.
נרצה לעשות הערכה, כלומר נחליף את כל המופעים של i בתוך e בערך 6 ולכן: $e[v/i] = \{*\ \ 6\ 6\}$.

<u>דוגמה 2</u>: נניח שיש לנו את : $\{with\ \{x\ 6\}\ \{*\ \ 7\ 8\}\}$, במקרה הזה, $i = none,\ e = \{*\ \ 7\ 8\},\ v = 6$.
נרצה לעשות הערכה, כלומר נחליף את כל המופעים של i בתוך e בערך 6 ולכן: $e[v/i] = \{*\ \ 7\ 8\}$.

<u>דוגמה 3</u>: נניח שיש לנו את : $\{with\ \{x\ 6\}\ \{+\ \ x\ \{with\ \{x\ 3\}\ 10\ \}\}\}$,
במקרה הזה, $i = x,\ e = \{+\ \ x\ \{with\ \{x\ 3\}\ 10\ \}\},\ v = 6$. נרצה לעשות הערכה, כלומר נחליף את כל המופעים של i בתוך e בערך 6 ולכן: $e[v/i] = \{+\ \ 6\ \{with\ \{6\ \ 3\}\ 10\ \}\}$.
נשים לב שבדוגמה הזאת ההגדרה שלנו לקויה כי ה-x הפנימי לא מוגדר נכון.

<u>הגדרה 1</u> - מופע של מזהה שבו מכריזים על משתנה חדש - נקרא **binding instance**.

<u>הגדרה 2</u> - המקום שבו המזהה מוגדר בגוף של ה-with (בבלוק) נקרא - **Scope**. למשל אם נסתכל למטה על ה-x הירוק, אז ה-scope שלו זה ה-x האדום (כלומר ה-x=6 רלוונטי ב-scope הזה).

<u>הגדרה 3</u> - שם מזהה שנמצא בתוך הגוף של ה-with אבל הוא לא binding - נקרא **bound instance**.

<u>הגדרה 4</u> - אם נסתכל על ביטוי מסוים, מזהה שאינו bounding ולא binding הוא נקרא **free instance**.

{ with { **x** 6 } { + **x** { with { **x** 3} 10 } } }
e = { + **x** { with { **x** 3} 10 } }

תיקון הגדרה פורמלית (הגדרה גם שגויה):

נסמן ב- $e[v/i]$ שתפקידו: מחליף (substitute) את כל המופעים של i בתוך E **שהם לא binding** עם v.

דוגמה - $\{with\ \{x\ 6\}\ \{+\ \ x\ \{with\ \{x\ 3\}\ x\}\}\ \ ==>\ \ i\ =\ x,\ e\ =\ \{+\ \ x\ \{with\ \{x\ 3\}\ x\}\},\ v\ =\ 6$

$$e[v/i]\ =\ \{+\ \ 6\ \{with\ \{x\ 3\}\ 6\ \}\ \}$$

קיבלנו שוב פעם תקלה, כי הצבנו 6 ב-x השני אבל אנחנו אמורים להציב 3.

תיקון הגדרה פורמלית (הגדרה גם שגויה):

נסמן ב- $e[v/i]$ שתפקידו: מחליף (substitute) את כל המופעים של i בתוך E שהם לא binding וגם **שהם לא בתוך סקופ מקונן** אז נחליף עם v.

דוגמה - $\{with\ \{x\ 6\}\ \{+\ \ x\ \{with\ \{x\ 3\}\ x\}\}\ \ ==>\ \ i\ =\ x,\ e\ =\ \{+\ \ x\ \{with\ \{x\ 3\}\ x\}\},\ v\ =\ 6$

$$e[v/i]\ =\ \{+\ \ 6\ \{with\ \{x\ 3\}\ x\ \}\ \}$$

עכשיו הכל תקין.

דוגמה נוספת:

$$\{with\ \{x\ 6\}\ \{+\ \ x\ \{with\ \{y\ 3\}\ x\}\}\ \ ==>\ \ i\ =\ x,\ e\ =\ \{+\ \ x\ \{with\ \{y\ 3\}\ x\}\},\ v\ =\ 6$$

$$e[v/i]\ =\ \{+\ \ 6\ \{with\ \{y\ 3\}\ x\ \}\ \}$$

אנחנו בבעיה שוב, כי אנחנו אמורים להציב ב-x את 6 אבל לפי ההגדרה שלנו זה לא ניתן לחישוב.

תיקון הגדרה פורמלית (הגדרה נכונה):

נסמן ב- $e[v/i]$ שתפקידו: מחליף (substitute) את כל המופעים של i בתוך E **שהם free** בביטוי v.

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

נניח שיש לנו $(eval\ (With\ 'x\ (Add\ (Num\ 5)\ (Num\ 3))\ (Mul\ (Id\ 'x)\ (Id\ 'x)))\ )$ אז השלבים יהיו:

1. $v \leftarrow eval(\ (Add\ (Num\ 5)\ (Num\ 3))\ )\ ==>\ 8$
2. $E2' \leftarrow subst(\ (Mul\ (Id\ 'x)\ (Id\ 'x)\ )\ 'x\ (Num\ v))\ ==>\ (Mul\ (Num\ 8)\ (Num\ 8)\ )$
3. $eval(E2')\ =>\ 64$

נרצה להפעיל את ה-parser כדי להפעיל את ה-eval שאנחנו רוצים לבנות, נזכר שפעמים קודמות אמרנו שאנחנו תחילה מקבלים string כלומר קוד שמיוצג במחרוזת טקסט פשוט ונרצה לבצע פעולה שתעביר אותנו לעולם החישוב שבנינו עבור השפה ובפרט עבור הפעולה with שנרצה לממש בשלמותו.

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

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

פונקציות

היינו רוצים לתת אופציה למתכנת לבוא ולאפשר למתכנת להתעסק עם פונקציות. לדוגמה פונקציה שבהינתן x היא תחזיר לנו x^2. נגדיר כעת פונקציות לשפה שלנו.

לדוגמה:

```
{ call {fun {x}
     {* x x}} 2}
```

האם אנחנו צריכים להוסיף אופציה לתת שם מסוים לכל פונקציה? - קיימת לנו את האופציה של With.
אז אם נגדיר את ה-fun כאובייקט מסויים אז בעצם ניתן להפעיל את with על ה-fun הזה.

```
{ with {f {fun {x}
     {* x x}}} {+ {call f 2} {call f 3}}}
```

אנחנו נרצה להוסיף את האופציה הזאת (פונקציות) בצורה הכללית והלא מוגדרת שהוצגה כאן.

למה כדאי בכלל להוסיף פונקציות לשפת תכנות?

1. כפילות קוד - למנוע חזרה על קוד.
2. היכולת להביע את עצמי באמצעות הקוד שלי (modularity), כלומר לתת שם לפונקציה עם תכלית.

היסטוריה על פונקציות

1. *פונקציות First Order* - זה הפונקציות הראשונות שהגדירו בתחילת ההיסטוריה של התכנות בעולם. למה לא לאפשר למתכנת לקחת קטע קוד, לשמור אותו בזיכרון ולתת לו שם מסוים?.
2. *פונקציות Higher Order* - לאחר מכן, אנשים הבינו שפונקציה לא אמורה לקבל כפרמטר סוגים שונים, למה לא לקבל פונקציה סוג x ולהחזיר סוג אחר של פונקציה y. (כמו למשל ב-Python).
3. *פונקציות First Class* - לאחר מכן אמרו - אם כבר אנחנו יודעים לקבל כפרמטר פונקציה ולהחזיר כפלט פונקציה אחרת אז למה לא להתייחס לפונקציה כאובייקט בפני עצמו? - למה לא להגדיר אותו כטיפוס כלשהו? למשל כמו int, double, string ואז אפשר יהיה לעשות פעולות על הפונקציות האלה.

חסרונות

- בכל שורה בשפה אנחנו יכולים להגדיר פעולה אחת בלבד.
- כל פלט של קוד צריך לתת שם.

נציג דוגמה לחסרון בקוד שבכל שורה יש רק פעולה אחת:

```
a = a*a
b = b*b
c = a + b
d = sqrt(c)
```

נשים לב שקשה להבין מה בדיוק אנחנו רוצים לחשב בקטע הקוד הזה מכיוון שאנחנו מבצעים בכל שורה רק פעולה אחת בלבד. כעת נציג את אותו החישוב בשורה אחת ונשים לב שאנחנו מבינים יותר את מטרת החישוב.

```
c = sqrt(a*a + b*b)
```

השפה FLANG

אנחנו נרצה לבנות שפת תכנות עם פונקציות ולכן נשנה את השם WAE ל-FLANG ונצרף אליו פעולות חדשות.
כלומר, אנחנו נחליף בקוד כל מופע של WAE ל- FLANG ונצרף אליה את **call** ואת **fun**:

```
<FLANG> ::= <num> 1
      | { + <FLANG> <FLANG> } 2
      | { - <FLANG> <FLANG> } 3
      | { * <FLANG> <FLANG> } 4
      | { / <FLANG> <FLANG> } 5
      | { with {<id> <FLANG>} <FLANG>} 6
      | <id> 7
      | { fun {<id>} <FLANG> } 8
      | { call <FLANG> <FLANG> } 9
```

הסבר:

- בירוק - זה יכול להיות כל ערך חוקי (למשל מספר כלשהו).
- באדום - יכול להיות כל FLANG למשל פונקציה כלשהי או `{* x x}` או כל סוג אחר של FLANG.
- בכתום - אמור להיות פונקציה כלשהי, לכן זה FLANG כי זה יכול להיות פונקציה מסוג fun או פונקציה אחרת מוכרת כלשהי (למשל sqt) שניתן לייצג אותה באמצעות id.
- בסגול - זה "הערך של הפונקציה", זה יכול להיות מספר או פונקציה אחרת שתחזיר ערך ולכן FLANG.

נגדיר את התוספים החדשים בשפה:

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

נזכר בשלבים שצריך לעבור כדי ליצור שפת תכנות:

1. מגדירים BNF - הגדרת הדקדוק של השפה.
2. כל String צריך להמיר ל-Sexpr.
3. ממירים את Sexpr לעץ הסינטקס האבסטרקטי לפי ה-Parser שעשינו ל-WAE.
4. עושים Eval של-WAE לסוג Number.

כעת נממש את שלב (3) - נשנה את ה-Parser, אנחנו צריכים להתייחס ל-parse-sexpr ולהוסיף את הפעולות החדשות לשפה בהתאם.

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

            [(cons 'fun more)
             ;; go in here for all sexpr that begin with a 'fun
             (match sexpr
               [(list 'fun (list (symbol: name)) body)
                (Fun name (parse-sexpr body))]
               [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]
              )]

            [(list 'call fun-expr arg-expr)
                    (Call (parse-sexpr fun-expr) (parse-sexpr arg-expr))]

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

השם more זה כללי אפשר במקומו "_", וזה פשוט רשימה כלשהי (כל רשימה מכל סוג).

נבצע מספר טסטים:

```
(test (parse "3") => (Num 3))
(test (parse "{ + 3  4 }") => (Add (Num 3) (Num 4)) )
(test (parse "{ + {- 3 2}  4 }") => (Add (Sub (Num 3)(Num 2))(Num 4)))
(test (parse "{+ 1 2 3 4}") =error> "bad syntax")
(test (parse "{fun {x} x}") => (Fun 'x (Id 'x)))
(test (parse "{fun {x} {/ x 5}}") => (Fun 'x (Div (Id 'x) (Num 5))))
(test (parse "{call   {fun {x} {/ x 5}}   8}") =>
                      (Call (Fun 'x (Div (Id 'x) (Num 5))) (Num 8)))
(test (parse "{fun x x}") =error> "bad syntax")
(test (parse "{with {sqr {fun {x} {* x x}}} {+ {call sqr 5} {call sqr 8}}}")
       => (With 'sqr
                (Fun 'x (Mul (Id 'x) (Id 'x)) )
                (Add (Call (Id 'sqr) (Num 5)) (Call (Id 'sqr) (Num 8)) )))
```

מימוש ה-subst עבור FLANG

כעת צריך לממש את השלב הבא - מימוש ה-subst כמו שעשינו עם השפה WAE. נזכר בכללים שהיו ל-subst,

כלומר לכל ערך x נרצה להציב במקומו את המספר הטבעי v לטובת חישוב בשלב הבא של ה-eval:

```
1.  {+ E1 E2}[v\x]          = {+ E1[v\x] E2[v\x]}
2.  {- E1 E2}[v\x]          = {- E1[v\x] E2[v\x]}
3.  {* E1 E2}[v\x]          = {* E1[v\x] E2[v\x]}
4.  {/ E1 E2}[v\x]          = {/ E1[v\x] E2[v\x]}
5.  y[v\x]                  = y
6.  x[v\x]                  = v
7.  {with {y E1} E2}[v\x]   = {with {y E1[v\x]} E2[v\x]}
8.  {with {x E1} E2}[v\x]   = {with {x E1[v\x]} E2}
9.  {call E1 E2}[v\x]       = {call E1[v\x] E2[v\x]}
10. {fun {y} E}[v\x]        = {fun {y} E[v\x]}
11. {fun {x} E}[v\x]        = {fun {x} E}
```

כעת לאחר שהצגנו הכנה לחישוב נכון, נממש את subst:

```
(: subst : FLANG Symbol FLANG -> FLANG)

(define (subst expr from to)
  (cases expr
    [(Num  n) expr]
    [(Add  l r) (Add (subst l from to) (subst r from to))]
    [(Sub  l r) (Sub (subst l from to) (subst r from to))]
    [(Mul  l r) (Mul (subst l from to) (subst r from to))]
    [(Div  l r) (Div (subst l from to) (subst r from to))]
    [(With name named body) (With name
                                  (subst named from to)
                                  (if (eq? name from)
                                      body
                                      (subst body from to)))]
    [(Fun name body)(Fun name (if (eq? name from) body
                              (subst body from to)))]
    [(Call fun-expr arg-expr) (Call (subst fun-expr from to)
                                    (subst arg-expr from to))]
    [(Id   name) (if (eq? name from) to expr)]))
```

אנחנו אפשרנו פונקציות אנונימיות ואפשרנו שם מיקומי על ידי with.
בשבוע שעבר הוספנו את העניין של הפונקציות לדקדקוד שלנו ולאחר מכן הוספנו אותו לעץ סינטקס אבסטרקטי שלנו ולאחר מכן הכנסנו אותו ל-parser שלנו.
כעת אנחנו נעבור על ה-eval.

ספציפיקציה פורמלית של ה-eval

1. $eval(N) = N$
2. $eval(\{+ \ E1\ E2\}) = eval(E1) + eval(E2)$
3. $eval(\{- \ E1\ E2\}) = eval(E1) - eval(E2)$
4. $eval(\{* \ E1\ E2\}) = eval(E1) * eval(E2)$
5. $eval(\{/ \ E1\ E2\}) = eval(E1) / eval(E2)$
6. $eval(id) = error$
7. $eval(\{with\ \{x\ E1\ E2\}\}) = eval(E2[eval(E1)\ change\ to\ x])$
8. $eval(\{fun\ \{x\}\ E\}) = \{fun\ \{x\}\ E\}$
9. $eval(\{call\ E1\ E2\}) = if\ \{fun\ \{x\}\ Ef\} \leftarrow eval(E1):$
   $then:\ eval(Ef\ [eval(E2)\ change\ with\ x])$
   $else:\ error!$

בגלל שהוא מוחזר מ-Parser זה לא הגיוני שזה גם יחזור מ-eval.
אנחנו נשתמש ב-FLANG בשביל לקיים את ה-eval.
המטרה של ה-eval היא להחזיר 2 סוגים:

1. מספר Number
2. ענן - Functions

נשים לב שיש לנו כאן בעיה, אנחנו לא יכולים להחזיר חישוב של מספר בכל מימוש פעולה מתמטית כמו למשל חיבור או חיסור כיוון שאנחנו נגדיר את ה-eval לקבל ולהחזיר FLANG אבל זה לא סוג של מספר ולכן נקבל שגיאה כי אי אפשר לבצע פעולות אריתמטיות ללא מספרים.
לכן, אנחנו נרצה לכתוב פונקציית עזר חדשה $arith - op$ שתקבל את החישוב ותבצע המרה למספר.

```
(: arith-op : ((Number Number) -> Number) FLANG FLANG -> FLANG)
(define (arith-op op arg1 arg2)
      (:Num->Number : FLANG -> Number)
      (define (Num->Number arg)
           (cases arg
                [(Num n) n]
                [else (error 'arith-op "expects a number, got: ~s"
                arg)]
           )
      )
     (Num (op (Num->Number arg1) (Num->Number arg2)))
)

( : eval : FLANG -> FLANG)
(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 name named-expr body)
               (eval (subst body
                            name
                            (eval named-expr)))]
          [(Id name) (error 'eval "free instance: ~s" name)]
          [(Fun name body) expr]
          [(Call (Fun name body) arg-exp)
           (let ([fval (eval fun-expr)])
                (cases fval
                 [(Fun name body) (eval (subst body
                                              name
                                              (eval arg-exp)))]
                )
           )
           (eval (subst body
                        name
                        (eval arg-exp)))]
     )

)
```

נבצע בדיקות:

```
(test (eval (Call (Fun 'x (Mul (Id 'x) (Num 4))) (Num 5)))=>(Num 12))

(test (eval (Call (With 'foo
                          (Fun 'x (Mul (Id 'x) (Num 4)))
                          (Id 'foo))
                     (Num 3)))
                => (Num 12))
```

נממש את ה-run.

```
(:run : String -> Number)
(define (run code)
      (let ([res (eval (parse code))])
      (cases res
            [(Num n) n]
            [else (error 'run "eval returned a non-number: ~s" res)]
      )
)
```

נבצע בדיקות:

```
(test (run "{call {with {foo {fun {x} {* x 4}}} foo} 3}") => 12)
(test (run "{with {sqr {fun {x} {* x x}}}
                        {+ {call sqr 5}
                           {call sqr 6}
                        }}") => 61)
(test (run "{call 4 3}") =error> "eval: expects a function, got:")
(test (run "{with {sqr {fun {x} {* x x}}} sqr}")
                =error> "run: eval returned a non-number")
```

Let vs. Lambda

בשפת Racket, ה-Let הוא **syntactic sugar** עבור קריאה ל-lambda (פונקציה אנונימית).

המשמעות של syntactic sugar זה לקחת קוד לא קריא ולהמציא איזושהי מילה שמורה שמטרתה לפשט את הביטוי המסורבל. כלומר "למתק" את הסינטקס.

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

זה יהיה שקול ל:

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

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

נשים לב שניתן להקביל את With ו-Fun & Call ל-Let ו-lambda.

כלומר, With הוא syntactic sugar עבור Fun & Call.

מודל דינמי

עד עכשיו התעסקנו במודל הסטטי שהוא מודל החלפות.
Substitution Caches הוא מודל דינמי.

- ה-Identifier יכול להיות למשל שם מזהה של ערך או של פונקציה.

מודל סטטי - Static Scope \ Lexical Scope

בשפות כאלו, הערך של כל identifier תלוי ב-scope שבו ה-identifier הוגדר.
כלומר אם נשנה את ה-identifier במהלך התוכנית אז היא לא באמת תשתנה כי הוא קבוע.

מודל דינמי - Dynamic Scope

בשפות כאלו, הערך של כל identifier תלוי ב-scope שבו משתמשים ב-identifier זה.
אם הגדרנו את ה-id להיות 3 ואז שינינו אותו למספר 5 אז הוא באמת ישנה - בשונה מהמודל הסטטי.

דוגמה - פונקציית עצרת

```
(define fact (lambda (n)
       (if (zero? n)
            1
            (* n (fact (- n 1)))
        )
      )
)

(let ([* +])
     (fact 5)
 )
```

מה תהיה התוצאה ב-2 המודלים (דינמי / סטטי)?

- במודל סטטי - ה-id משומש רק ב-scope שבו הוא מוגדר. במקרה זה, ה-* ב-let משתנה ל-+ אבל זה לא ישפיע על ה-fact כיוון שהם לא נמצאים באותו ה-scope. כלומר, ה-* ב-fact נשאר כפל כמו שהוא. לכן, התוצאה במודל זה היא 120.
- במודל דינמי - במקרה זה, השינוי של * ב-let ל-+ באמת משפיע על החישוב של fact. כלומר ה-* בפונקציה של fact תשתנה ל-+ ולכן נחשב $5 + 4 + 3 + 2 + 1 + 1 = 16$.

חלאס.