Assignment 2 part 1: Theoretical part

1. דוגמאות לערכים וביטויים מסוגים שונים:
   א) הביטוי הבוליאני של true: #t.
   ב) ביטוי של משתנה: x.
   ג) ביטוי מסוג app expression: ((lambda (x) x + 1) 3).
   ד) הערך של חישוב הביטוי 7.
   ה) ה-symbol שנוצר מהביטוי 'abpurple.
   ו) ה-closure שנוצר מחישוב הביטוי (lambda(x) x).

2. Special form הוא ביטוי שמחושב בצורה השונה מהכלל הגנרי לחישוב ביטויים, שהוא באופן רקורסיבי חישוב כל תתי הביטויים והפעלת תת הביטוי הראשון על כלל תתי הביטויים שבאים אחריו.
   דוגמה: הביטוי (define x 7) הוא special form.

3. ההגדרה free variable (משתנה חופשי) מתייחסת למשתנה אשר מופיע בצורת varRef ואינו קשור לאף varDecl בקוד, כלומר אין לו הכרזה שקושרת אותו.
   דוגמה: בביטוי (lambda (x) x + y) המשתנה y מופיע חופשי מכיוון שאין הכרזה חוקית שקושרת אותו.

4. S-expression הנקרא גם datum הוא ביטוי string בשפת scheme, הדומה בתפקידו ל-json ב-javascript, במובן זה שניתן לקודד בו ולקרוא ממנו מידע של השפה. ניתן להפוך כל ביטוי ב-scheme ל-s-exp באמצעות סימון " ' " (גרש בודד) בתחילתו.
   דוגמה: הביטוי '(+ 1 2) הוא ביטוי s-exp.

5. Syntactic abbreviation או syntactic sugar הוא ביטוי חוקי בשפה אשר מאפשר לכתוב פעולה הקיימת בשפה, בדרך נוספת שזהה סמנטית, אך פשוטה יותר לכתיבה והבנה של המתכנת.
   דוגמה:
   א) הביטויים (let (x 7)) x) ו-((lambda (x) x) 7).
   ב) הביטויים ((if (< x 7) (+ x 7) (- x 7)) ו-((cond ((< x 7) (+ x 7)) (else (- x 7)).

   בשני המקרים קיים שוויון סמנטי בין שני הביטויים אולם סינטקטית הביטוי הראשון קל יותר לכתיבה וקריאה.

6. נוכל לבצע כל ב-L30 כל פעולה שניתן היה לבצע ב-L3 מכיוון שניתן ליצור רשימות ב-scheme גם לא בתור literal וללא המילה השמורה list, באמצעות הרשימה הריקה באופן הבא: ((cons 1 (cons 2 (cons 3 '())) = '(1 2 3). לכן כל תכנית שנדרש בה השימוש ברשימה תוכל ליצור אותה באופן הזה, וזהו השינוי היחיד שקרה בשפה.

7. הגרסה של $PrimeOp$ יעילה יותר ברוב המקרים מכיוון שהיא אינה דורשת חיפוש בסביבה בשביל חישוב של פעולות פרימיטיביות, ואנו צופים שיהיו הרבה חישובים כאלה במהלך התוכנית.

   הגרסה של $Closure$ או $PrimeProc$ היא גנרית יותר וניתנת להפעלה ללא תיפול במקרים ספציפיים, עקרונות שיאפשרו שליטה טובה יותר בפעולת התוכנית.

8. נתבונן בפעולה של פונקציות שונות שלמדנו על מערך מהתחלה לסוף ומהסוף להתחלה:
   א. Map – ניתן להפעיל את הפונקציה משני הכיוונים באופן זהה מכיוון שהפעולה מתבצעת על כל איבר בנפרד.
   ב. Filter – ניתן להפעיל את הפונקציה משני הכיוונים באופן זהה מכיוון שהבדיקה של האיברים מתבצעת על כל איבר בנפרד.
   ג. Reduce – לא תמיד הפעולה מתבצעת באופן זהה בהפעלה משני הכיוונים. לדוגמא בהפעלת פעולת חיסור על רשימה בת 2 מספרים, הפעלה מימין לשמאל תחסר את המספר השני מהראשון, ואילו הפעלה הפוכה תחסר את הראשון מהשני, וברור כי התוצאה עשויה להיות שונה בשני המקרים.

ד. Compose – לא תמיד הפעולה מתבצעת באופן זהה בהפעלה משני הכיוונים. לדוגמא עבור הפונקציות x^2 ו-$2x$, אבל $x^2 \circ 2x \neq 2x \circ x^2$, כלומר ההרכבה לא זהה בשני המקרים.

Assignment 2 part 2: scheme functions contracts

1. *Last element*
   - Signature: last-element(lst)
   - Purpose: returns the last element of lst
   - Type: [list(T) -> T]
   - Example: (last-element '(1 2 3)) should return 3
   - Pre-conditions: empty? lst == #f
   - Post-condition: result is of type T
   - Tests: (last-element '(1 2 3)) ==> 3

2. *Power*
   - Signature: power(n1, n2)
   - Purpose: calculate n1 to the power of n2
   - Type: [Number*Number -> Number]
   - Example: (power 2 3) should produce 2^3=8
   - Pre-conditions: n1 > 0 && n2 >= 0 && n2 is natural
   - Post-condition: result >= 0
   - Tests: (power 2 3) ==> 8

3. *Sum list power*
   - Signature: sum-lst-power(lst, n)
   - Purpose: calculate the sum of all the elements in lst to the power of n
   - Type: [list(Number)*Number -> Number]
   - Example: (sum-lst-power '(1 2 3) 2) should produce 1^2+2^2+3^2=13
   - Pre-conditions: n >= 0 && n is natural
   - Post-condition: result >= 0
   - Tests: (sum-lst-power '(1 2 3) 2) ==> 13

4. *Number* from digits
   - Signature: num-from-digits(lst)
   - Purpose: calculates the decimal number the list represents
   - Type: [list(Number) -> Number]
   - Example: (num-from-digits '(3 1 4)) should produce 314
   - Pre-conditions: empty? lst = #f && lst elements are natural
   - Post-condition: result is natural
   - Tests: (num-from-digits '(3 1 4)) ==> 314

5. *Calculate number from digits*
   - Signature: calc-num-from-digits(lst, acc)
   - Purpose: calculates the decimal number the list represents and appends it at the end of the given acc
   - Type: [list(Number)*Number -> Number]
   - Example: (calc-num-from-digits '(3 1 4) 1) should produce 1314
   - Pre-conditions: empty? lst = #f && acc is natural
   - Post-condition: result is natural
   - Tests: (calc-num-from-digits '(3 1 4) 1) ==> 1314

6. *Is narcissistic*
   - Signature: is-narcissistic(lst)
   - Purpose: checks if a list of numbers is a narcissistic list
   - Type: [list -> Boolean]
   - Example: (is-narcissistic '(1 5 3))== #t
   - Pre-conditions: lst is a list of numbers
   - Post-condition: result is Boolean
   - Tests: (is-narcissistic '(1 2 3)) ==> #f, (is-narcissistic '(1 5 3)) ==> #t

7. *List Size*
   - Signature: lst-size(lst)
   - Purpose: calculate the size of a list
   - Type: [list(T) -> Number]
   - Example: lst-size '(1 4 5) = 3
   - Pre-conditions: none
   - Post-condition: result >=0 && result is a natural number
   - Tests: (lst-size '(#t #f)) ==> 2, (lst-size '())=0

8. *Empty?*
   - Signature: empty?(lst)
   - Purpose: checks if a list is empty
   - Type: [list(T) -> Boolean]
   - Example: (empty? '(1 2 3)) should return #f
   - Pre-conditions: none
   - Post-condition: result should be a Boolean
   - Tests: (empty? '(1 2 3)) ==> #f, (empty? '()) ==> #t