עקרונות שפות תכנות
מני אדלר

2021

חובות הקורס: 5 עבודות בית (25%), מבחן סופי (75%)

שעות קבלה (בתאום מראש): יום חמישי 10-12

---

1. מבוא, תחביר סמנטיקה וטיפוסים ב TypeScriptתבנית

1.1 מבוא

1.1.1 מטרות ונושאי הקורס

ספר עזר לרופא שיניים בהוצאת כתר

בלשנים:

- חוקי השפה: פונטיקה, מורפולוגיה, תחביר, משמעות,...
- אפיון השפות השונות בעולם ע"פ מאפייניהן, משפחות של שפות, וכו'
- התפתחות השפה, אבולוציה

← מטרה ראשית בקורס: ברצוננו להיות בלשנים של שפות תכנות
- מהם הרכיבים השונים של שפת תכנות (בפרט, תחביר וסמנטיקה/משמעות)
- סוגים שונים של שפות תכנות
- התפתחות של שפת תכנות

← מטרות משנה:

- הכרות עם שפות תכנות חדשות
  - שפות פונקציונאליות: JavaScript/TypeScript, שפות ש'נמציא' L1-L7
  - שפה לוגית: Prolog / שתי שפות לוגיות שנגדיר.

- עיצוב מטה-תוכניות (Meta-Programming)

תוכניות המקבלות כקלט תוכניות אחרות, ועושות איתן משהו:
  - אינטרפרטר: מקבל תוכנית, מריץ/מחשב אותה.
  - בדיקת טיפוסים (type checking): מקבלת תוכנית עם טיפוסים ובודקת את תאימותם (נעשה בד"כ כחלק מתהליך הקומפילציה)

- מערכת הסק טיפוסים: מקבלת תוכנית ללא טיפוסים, מחזירה את התוכנית מהקלט עם טיפוסים.
- המרת קוד: קבלת תוכנית בשפה אחת, והחזרת תוכנית שקולה בשפה אחרת. בפרט, הקומפיילר מקבל תוכנית בשפה גבוהה (C++, Java) ומתרגם אותה לתוכנית בשפה נמוכה יותר (Assembly, Byte code)
- אימות קוד: קבלת תוכנית, ובדיקה האם היא תקינה.

תוכנית הקורס:

1. מבוא, תכנות פונקציונאלי ב TypeScript, טיפוסים
2. סמנטיקה אופרציונאלית: הרכיבים המרכזיים של שפות התכנות - המבנה והמשמעות. הגדרת השפות L1-L4
3. הסק טיפוסים (מימוש מערכת), השפה L5
4. אבסטרקציה של בקרת הריצה, השפות L6-L7
5. תכנות לוגי, שפה לוגית, אינטרפרטר וכו'

1.1.2 דגמי שפות תכנות

- שפות אימפרטיביות (Imperative Languages)
  תוכנית = רצף של פקודות
  הרצת תוכנית = ביצוע של הפקודות, בזו אחר זו
  דוגמאות: Java, C++, Python

- שפות הצהרתיות (Declarative Languages)
  תוכנית = הצהרה על הדבר המבוקש
  דוגמא: SQL, התוכנית היא שאילתא בה אנחנו מצהירים מה אנו מעוניינים לקבל

- שפות מבניות (Structural Languages)
  תחביר השפה כולל מבנים מקוננים (כך שאין צורך ב goto): if-else, do-while
  דוגמאות: Python ,Java, C++

- שפות פרוצדוראליות (Procedural Languages)
  תחביר השפה מאפשר להגדיר קוד כפרוצדורה, כך שניתן לקרוא ממקומות שונים בקוד.
  דוגמאות: Python ,Java, C++

- שפות פונקציונאליות (Functional Languages)

  - התוכנית היא <u>ביטוי</u>, או סדרת ביטויים, ולא רצף של פקודות
    הרצת התוכנית היא <u>חישוב של הביטוי</u>(ים), כלומר מציאת <u>הערך</u> שלו, ולא ביצוע של הפקודות

    דוגמא:

    (2 * 3) + (4 * 5)

  - פונקציות הן גם כן ביטויים. קריאה לפונקציה היא חישוב של ביטוי.

דוגמא:

```
function square(x) {
  return x*x;
}

square(3);
```

אפשר להעביר פונקציה כפרמטר לפונקציה אחרת, אפשר להחזיר פונקציה כערך מפונקציה אחרת.

- פונקציה מקבלת פרמטרים ומחזירה ערך: אין פעולת השמה, ובמובן הרחב יותר אין side-effects

- פונקציות מסדר גבוה
  מאחר שגם פונקציה היא ביטוי, ניתן להעביר פונקציה כפרמטר לפרוצדורה אחרת, וכן להחזיר כערך מפרוצדורה פונקציה.

יתרונות התכנות הפונקציונאלי:
- אימות קוד
- מקבול
- הפשטה/עיצוב

דוגמא: L1-L7, Scheme, JS, Python

- שפות מונחות עצמים (Object-Oriented)

תוכנית = סדרת העברת הודעות (קריאה למתודות) בין אובייקטים.
דוגמאות: Java, C++

- שפות מונחות אירועים (Event-driven)
  תוכנית = הגדרת תגובות שונות לאירועים שונים

דוגמא: JavaScript Node

- שפות לוגיות (Logic Programming)
  תוכנית = אוסף של עובדות, כללי הסק, ושאילתא
  הרצת התוכנית הינה ביצוע של שאילתות על העובדות ועל ההיסקים

1.1.3 התפתחות היסטורית של שפה

נקודת פתיחה: שפה אימפרטיבית, כלומר ניתן להגדיר אוסף פקודות

```
console.log(0*0);
console.log(1*1);
console.log(2*2);
```

```
console.log(3*3);
```

חסרונות:

- חזרתיות שלא לצורך (בפרט, מספר רב של פקודות דומות)
- התאמה לערכים אחרים (נניח במקום 0-3, 4-7) דורשת שינוי של הקוד
- תחביר השפה לא מלמד על כך שמדובר בדפוס חוזר

← שפה עם מבנה:

- משתנים
- מבנה נתונים: רשימה
- לולאה

```
let number = [0,1,2,3];
for (let i=0; i< numbers.length; i++)
   console.log(numbers[i] * numbers[i]);
```

חסרונות:

- שינוי הפרמטרים (נניח ל 4-7) דורש שינוי של הקוד (המשתנה numbers), כלומר יש להפריד את ה'לוגיקה' מהנתונים

← שפה פרוצדורלית

```
function printSquares(numbers) {
   for (let i=0; i< numbers.length; i++)
      console.log(numbers[i] * numbers[i]);
}

printSquares([0,1,2,3]);
printSquares])4,5,6,7;([;
```

חסרון: הלוגיקה מורכבת מדי. הפרוצדורה printSquares מבצעת שלושה דברים - העלאה בריבוע, הדפסה, מעבר על אברי המערך.
מקשה על אימות נכונות הקוד, ועוד.
נפריד את הלוגיקה בעזרת פרוצדורות נוספות:

```
function squares(numbers) {
   for (let i=0; i< arr.length; i++)
      numbers[i] = numbers[i] * numbers[i];
}

function print(arr) {
   for (let i=0; i< arr.length; i++)
      console.log(arr[i]);
}
```

```
function printSquares(numbers) {
      squares(numbers);
      print(numbers);
}

printSquares([0,1,2,3]);
printSquares])4,5,6,7;([;
```

חסרונות:

- יש מצב משותף שמתעדכן (מערך המספרים numbers בפרוצדורה squares): מקשה על מקביליות, אימות קוד, אופטימיזציות... ניתן היה לפתור זאת על ידי החזרת מערך חדש
- הפרוצדורות print ו squares מבצעות עדיין שני דברים – מעבר על מערך ופעולה על כל איבר.

← תכנות פונקציונאלי

```
function square(number) {
      return number * number;
}

function print(obj) {
  console.log(obj);
}

const square : (n : number) => number = (n : number) => (n * n);

function printSquares(numbers) {
     map(print, map(square,numbers));
}

printSquares([0,1,2,3]);
printSquares])4,5,6,7;([;
```

הערה: סוג כזה של פונקציות, המקבל פונקציה ונתונים ועושה איתם משהו מכונה **'פונקציות מסגר גבוה'**.

דוגמא נוספת: הפונקציה filter.
מקבלת – פונקציה בוליאנית, ורשימה.
מפעילה את הפונקציה הבוליאנית על כל אחד מאיברי הרשימה, ומחזירה את רשימת האיברים שצלחו את הפונקציה הבוליאנית (כלומר, שהפונקציה הבוליאנית החזירה עבורם ערך אמת).

```
function isEven(x) {
```

```
        return x % 2 == 0;
}

filter(isEven, [0,1,2,3]);
→ [0,2]
```

דוגמא נוספת (בתרגול): הפונקציה reduce

ניתן אף להרכיב שתי פונקציות לפונקציה מורכבת חדשה, בעזרת הפונקציה מסדר גבוה compose.
הפעולה compose(f,g) מחזירה את הפונקציה המורכבת f(g(x)) [עבור f g המקבלות פרמטר אחד x]

הערה: קריאה לפונקציות מסדר גבוה map, filter עם פרמטר אחד בלבד – הפונקציה לביצוע ללא רשימת האברים להפעלה – מחזירה פונקציה המקבלת רשימה ומבצעת על איבריה את הפונקציה שניתנה בשלב הראשון כפרמטר.
לדוגמא:
map(square) מחזירה פונקציה המקבלת רשימה ומפעילה על איבריה את הפונקציה square

```
let squareF = map(square);
squareF([0,1,2,3]);
→ [0,1,4,9]

let isEvenF = filter(isEven);
isEvenF([0,1,2,3]);
→ [0,2]
```

הרכבת הפונקציות בעזרת compose:

```
let isEvenSquareF = compose(isEvenF, squareF);
isEvenSquareF([0,1,2,3]);
→ [0,4]
```

<u>שקילות של פונקציות</u>

הרעיון הכללי: שתי פונקציות שקולות, אם עבור אותו קלט (פרמטרים) הן מחזירות את אותו ערך.

פורמלית:
נתונה פונקציה f עם תחום הגדרה D וטווח R
נתונה פונקציה g
הפונקציה f שקולה לפונקציה g אם"ם:

- התחום של g הוא גם D, והטווח של g הוא גם R

- לכל x ב D מתקיים: f(x) = g(x)

הגדרה זו תקפה לשפות תכנות פונקציונאליות, כי אין להן side-effect
אם כי, צריך לכלול גם תופעות של הרצת קוד:

- אם f(x) זורקת exception, אז גם g(x) זורקת exception
- אם f(x) לא מסתיימת, אז גם g(x) לא מסתיימת

לדוגמא:

f = compose(filter(isEven), map(cube))
g = compose(map(cube), filter(isEven))

האם f ו g שקולות?
יש להראות כי לכל מערך סופי $a=[a_1,...a_l]$ מתקיים $f(a) = g(a)$

ע"פ הגדרת compose:

$$\begin{aligned} f(a) &= filter(isEven, map(cube, a)) \\ &= filter(isEven, [cube(a_1), \ldots, cube(a_l)]) \end{aligned}$$

$$\begin{aligned} g(a) &= map(cube, filter(isEven, a)) \\ &= map(cube, filter(isEven, [a_1, ..., a_l])) \end{aligned}$$

על פי הגדרת filter ו map:

$$\begin{aligned} f(a) &= [cube(a_{i_1}), \ldots, cube(a_{i_k})] \\ &\quad |\forall j \in \{i_1, \ldots, i_k\}, isEven(cube(a_j)) \\ &\quad and \\ g(a) &= map(cube, [a_{j_1}, \ldots, a_{j_m}]) \\ &= [cube(a_{j_1}), \ldots, cube(a_{j_m})] \\ &\quad | \ \forall j \in \{j_1, \ldots, j_m\}, isEven(a_j) \end{aligned}$$

כך שנשאר רק להראות שמתקיים

1.2 תחביר, סמנטיקה, וטיפוסים ב TypeScript

1.2.1 סמנטיקה אופרציונאלית

1. תחביר וסמנטיקה

"אבנים שחקו מים" (איוב יד)
מי שחק את מי?

3 + 5 * 3
מה הערך של הביטוי?

התחביר מגדיר את המבנה של המשפט/הביטויים (בפרט, כדי למנוע דו משמעות)
הסמנטיקה מגדירה את המשמעות של המבנה – את משמעות המשפט או את ערך הביטוי

🡸 עבור כל שפת תכנות, יש להגדיר את המבנה והסמנטיקה של השפה. כלומר, מהם הביטויים החוקיים בשפה, וכיצד מחשבים כל ביטוי לכדי ערך.
הגדרה זו חשובה עבור מי שלומד השפה, עבור מי שכותב מטה-פרוגמינג המקבל כקלט תוכנית בשפה, וכן עבור מי שרוצה להוכיח שקילות בין תוכניות וכו'.

סמנטיקה פורמאלית, מגדירה את האופן שבו יש לפרש/לחשב ביטוי נתון כערך.
קיימות דרכים לתאר סמנטיקה באופן פורמאלי. בקורס, נלך בגישת הסמנטיקה האופרציונאלית:
המשמעות של ביטוי בשפת תוכנת מוגדרת על פי תהליך החישוב שלו מביטוי לערך:

- על ידי הגדרת סדרת המצבים שהביטוי עובר עד אשר הוא הופך לערך

```
Initial state: [x:5, y:3, z:7]
Program: "z = x; x = y; y = z;"

Step 1: execute "z=x;"
   Remaining program: "x=y; y=z;"
   New state: [x:5; y:3, z:5]
Step 2: execute "x=y;"
   Remaining program: "y=z;"
   New state: [x:3, y:3, z:5]
Step 3: execute "y=z;"
   Remaining program: ""
   New state: [x:3, y:5, z:5]
```

- על ידי תיאור תהליך החישוב
  נתון ביטוי E
  - זיהוי סוג הביטוי
  - זיהוי תתי הביטויים
  - חישוב תתי הביטויים בהתאם לסוג הביטוי הכולל אותם [פירוט של הסוגים השונים]
  - החזרת הערך של הביטוי E, על פי הערכים של תתי הביטויים שלו, ולאור סוג הביטוי

```
let x = 12
 (x > 7) ? (x * 3) : 9
```

2. ביטויים וערכים

כזכור, המונח ביטוי מתייחס למבנים השונים בשפת התכנות, והערכים הם המשמעות של ביטויים אלו.

ניתן לסווג את הביטויים והערכים השונים בשפה ע"פ שני מאפיינים:

- פרימיטיבי / לא פרימיטיבי
  האם הביטוי או הערך הינם חלק מובנה בשפה (פרימיטיב) או שהם הוגדרו ע"י המתכנתים.

  לדוגמא:
  ביטוי פרימיטיבי: 3, true, +
  ערך פרימיטיבי: הערך המספרי 3, ערך אמת, פעולת החיבור

  ביטוי שאינו פרימיטיבי: x
  ערך שאינו פרימיטיבי: ה enum RED

- אטומי / מורכב

  האם הביטוי או הערך הם פשוטים, כלומר לא כוללים בתוכם תתי ביטויים, או מורכבים, כלומר כוללים בתוכם תתי ביטויים.

  לדוגמא:
  ביטוי אטומי: 3, x
  ביטוי מורכב: 3 + 5, if x > 7 then 14 else 17

1.2.2 טיפוסים

1.2.2.1 טיפוס כקבוצה

נגדיר טיפוס כ<u>קבוצה של ערכים</u>
לדוגמא:
int - קבוצת המספרים השלמים
String – קבוצת כל המחרוזות האפשריות
boolean – הקבוצה {true, false}
any – קבוצת כל הערכים האפשריים

ניתן בשפות תכנות להגדיר טיפוסים חדשים על בסיס הטיפוסים הקיימים, ע"י פעולות על קבוצות:

- מכפלה קרטזית: לדוגמא, המחלקה 'קלף' הכוללת שלושה שדות – מספר, צבע, צורה – היא הקבוצה המתקבלת ע"י המכפלה הקרטזית של קבוצות המספרים {2-10, נסיך, מלכה, מלך, אס, ג'וקר}, הצבעים {שחור,אדום} והצורות {לב, תלתן, יהלום, עלה}
- איחוד של טיפוסים: יונקים וציפורים
- חיתוך של טיפוסים: יונקים שהם אוכלי עשב.
- נראה בהמשך הגדרת טיפוס חדש ע"י פעולת disjoint union

ניתן אף לאפיין קשרים בין טיפוסים, כיחסים בין קבוצות:

- <u>ספיציפיות</u> (טיפוס ותת-טיפוס): 1T (כלב) הוא תת-טיפוס של 2T (חיה) אם הקבוצה 1T (כל הכלבים האפשריים) מוכלת בקבוצה 2T (כל החיות האפשריות)
- <u>זרות</u>: הטיפוס 1T (יונקים) זר לטיפוס 2T (ציפורים), אם החיתוך של הקבוצות 1T ו 2T (ציפורים יונקות) ריק.

1.2.2.2 Type Checking

מנגנון (!meta-programming) המבטיח תאימות של טיפוסים בקוד התוכנית. ובפרט תאימות של טיפוסי ביטויים לטיפוסי ערכים.

int i = 'a'

- בשפות עם טיפוסים (כמו java) ניתן לבצע בדיקה זו בזמן קומפילציה.
- בשפות ללא טיפוסים (כמו JavaScript) הבדיקה תתבצע בזמן ריצה.

הערה: גם אם אין טיפוסים למשתנים בשפה, קיימים תמיד טיפוס לערך.
לדוגמא, בקוד הבא - אין ל-x טיפוס מוגדר, אך לאחר ההשמה הערך הנוכחי שלו הוא תו, כך שתהיה שגיאת זמן ריצה כאשר יחושב ביטוי ה if

```
x = 'a'
if (x > 7)...
```

כדאי להגדיר טיפוסים:
- בדיקת תאימות טיפוסים מראש (לא בזמן ריצה)
- הקוד קריא יותר
- תורם לעיצוב נכון יותר

1.2.2.3 טיפוסים ב TypeScript

- הרחבה של JavaScript עם אפשרות להגדרת טיפוסים
- הקומפיילר של TS:
  - מסיק טיפוסים
  - בודק את תאימות הטיפוסים
  - מוריד את הטיפוסים מהקוד, ומשאיר לאינטרפרטר של JS להריץ אותו.
- סוגי טיפוסים ב TypeScript
  - פשוטים (אטומי)
    - boolean – הקבוצה {true, false}
    - number – קבוצת כל המספרים (כמו float)
    - string – קבוצת כל המחרוזות
    - null – הקבוצה {null}
    - undefined – הקבוצה {undefined}
    - any – קבוצת כל הערכים האפשריים
  - מורכבים
    - מערכים, arrays
    - מילונים, maps

לדוגמא:

```
let arr = [1,2,3],
    map = { ‘a’ : 1, ‘b’ : true};
```

- ניתן 'לברר' בזמן ריצה מה הטיפוס של ביטוי בעזרת הפונקציה typeof (זה קצת מוגבל ב JS אך טוב יותר ב TS):

```
typeof (1)
→[number]
typeof(‘1’)
→[string]

typeof([1,2,3])
→[object]

typeof({ 'a' : 1, 'b' : true})
→[object]
```

ניתן לברר באופן מדויק יותר ב JS את הטיפוס ע"י השאלה instanceof

```
let arr = [1,2,3], map = { a: 1, b: true };
  console.log(arr instanceof Array);
  console.log(map instanceof Array);
[true, false]
```

<u>הגדרת ערכים בשפת התכנות</u>

כיצד מתארים בקוד התוכנית ערכים שונים? מהם הביטויים המתארים ערכים שונים?

לדוגמא:

- מספרים
  כיצד מתארים מספרים בשפה?
  3, הוא הביטוי למספר 3
  מספר רציונאלי: 0.5, ½
  מספר מדומה: ?

- מחרוזות
  'אבג', "אבג",...

- מערכים
  [1,2,3], {1,2,3},...

- טבלאות / מילונים
  {a:1, b:true}

ב Java לא קיימת דרך לתאר טבלה. צריך לעשות זאת בקוד:

```
Map<String,Object> m = new HashMap<String,Object>();
m.put("a",1);
m.put("b", true);
```

האופן שבו מוגדרים ערכים בעזרת ביטויים מכונה *literal expression*. או במילים אחרות, literal expression הוא ביטוי המתאר ערך.

ב JavaScript קיימת דרך כללית לתאר כל סוג של ערך שהוא, בעזרת הפורמט Java Script Object Notation (JSON)
ב JSON ניתן לתאר כל ערך מורכב באופן רקורסיבי (פרטים בתרגול)

לדוגמא: תיאור ערך של רשימת סרטים

```
let movieList = [
    {
      name: "New Releases",
      videos: [
        {
          "id": 70111470,
          "title": "Die Hard",
          "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
          "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
          "rating": 4.0,
          "bookmark": []
        },
        {
          "id": 654356453,
          "title": "Bad Boys",
          "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
          "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
          "rating": 5.0,
          "bookmark": [{ id: 432534, time: 65876586 }]
        }
      ]
    },
    {
      name: "Dramas",
      videos: [
        {
          "id": 65432445,
          "title": "The Chamber",
          "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
          "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
```

```
            "rating": 4.0,
            "bookmark": []
        },
        {
            "id": 675465,
            "title": "Fracture",
            "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
            "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
            "rating": 5.0,
            "bookmark": [{ id: 432534, time: 65876586 }]
        }
    ]
  }
];
```

<u>הגדרת טיפוסים ב TS</u>

טיפוסים פשוטים:

```
let x : number = 5,
  s : string = 'a',
  b : boolean = true;
  b
```

מערכים:

```
let arr : number[] = [1,2,3],
  x : number = arr[0];

  arr = [1,'b',4]   /* type error */
  x = true   /* type error */
```

מילונים

```
let s1 : { name : string, age : number} = { name : 'Danny', age : 3},
  s2 : { name : string, adult : boolean} = { name : 'Dinna', adult : true};
```

מבחינת קבוצות, הטיפוס של s1 הוא כל המילונים בעולם עם שדה בשם name מסוג מחרוזת ושדה בשם age מסוג מספר, בעוד שהטיפוס של s2 הוא כל המילונים בעולם עם שדה בשם name מסוג מחרוזת ושדה adult מסוג ערך בוליאני.
כלומר הטיפוסים של s1 ו s2 אינם תואמים.

מה לגבי:

```
s1 : { name : string, age :number }
s2 : { name : string, age :number, adult: boolean }
```

האם s1 תואם ל s2? להפך? בכלל לא?
הטיפוס של s1 הוא קבוצת כל המילונים שיש בהם (לפחות) שדה name מסוג מחרוזת ושדה age מסוג מספר.
הטיפוס של s2 הוא קבוצת כל המילונים שיש בהם שדה name מסוג מחרוזת, שדה age מסוג מספר, ושדה adult בוליאני.
← הטיפוס של s2 הוא תת-טיפוס / יותר ספציפי מהטיפוס של s1.

<u>מתן שם לטיפוס</u>

ראינו כי ניתן להגדיר טיפוסים חדשים. ניתן אף לתת שם, לשם נוחות, לטיפוסים אלה.

```
type <type name> = <type>
```

לדוגמא:

```
type StringArray = string[]
type Person = { name : string }
```

חסרון: ה type הינו סה"כ alias, כלומר חוסך מאיתנו להקליד שוב ושוב את הגדרת הטיפוס.
כחומר, הקומפיילר מחליף כל פעם את שם הטיפוס המוגדר במה שהוגדר עבורו.
← בעייתי עבור מבנים רקורסיביים (נקבל מחרוזת טיפוס אינסופית)

```
type Person = { name : string, kids : Person[] }
```

← נדרש מבנה טיפוס מיוחד עבור טיפוסים רקורסיביים

```
interface Person { name : string, kids : Person[] }
```

באופן זה ניתן גם להגדיר מילונים התלויים אחד בשני:

```
interface Course { id : number, name : string, students : Student[]}
interface Student { id : number, name : string, courses : Course[]}
```

<u>טיפוס לפונקציות</u>:

```
function add(x : number ,y : number ) : number { return x+y; }

const add : (x : number, y : number) => number =
      (x : number, y : number) : number => x+y;
add(1,2)
3
```

<u>שימוש בפעולות על קבוצות לשם הגדרת טיפוסים חדשים</u>:

- מכפלה קרטזית
  - הגדרת טיפוס של מילון לעיל

- איחוד וחיתוך

קבוצת המספרים והמחרוזות
```
type NumberOrString = number | string;
```

קבוצת המספרים והערכים {true,false}
```
type NumberOrBoolean = number | boolean;
```

קבוצת המספרים (חיתוך של שתי הקבוצות הנ"ל)
```
type Numbers = NumberOrString & NumberOrBoolean;
```

- בידול (disjoint union)

דוגמא:
```
interface Person { name : string, address : string }
interface Variable { name : string, address : string}

let p : Person = { name : 'yossi', address : 'beer sheva'},
    v : Variable = { name : 'x' , address : '14576'};
   v = p;
```

תאימות הטיפוסים ב JS מבוססת על מבניות (Structural) ועל ההשמה v=p חוקית.
הקבוצות של שני הטיפוסים זהות.
בניגוד לשפות כמו Java בהם התאימות הנדרשת מבוססת על שם הטיפוס (Nominal)

כיצד נוכל לאלץ ולגרום להשמה הנ"ל להיות לא חוקית גם בשפות עם Structural Checking?
← נגרום לכך שמבנה הטיפוסים יהיה שונה (באופן 'מלאכותי')
```
interface Person { tag : “person”[1], name : string, address : string }
interface Variable { tag : “variable”, name : string, address : string}

let p : Person = { tag : “person”, name : 'yossi', address : 'beer sheva'},
    v : Variable = { tag : “variable”, name : 'x' , address : '14576'};
   v = p;   // incorrect
```

דוגמא נוספת בה נדרש ל"בדל" טיפוסים: כאשר נדרש להבחין בין תתי טיפוסים של טיפוס כללי אחד.

```
type Shape = Circle | Rectangle | Triangle;

interface Circle {
   tag: "circle";
```

[1] כאשר יש ערך (כמו המחרזות “person” או המחרוזת "variable") במיקום של טיפוס, המשמעות היא הקבוצה המכילה את ערך זה בלבד: {“variable”}, {“preson”}

```
   center: {x:number, y:number};
   radius: number;
}

interface Rectangle {
   tag: "rectangle";
   upperLeft: {x:number, y:number};
   lowerRight: {x:number; y:number};
}

interface Triangle {
   tag: "triangle";
   p1: {x:number, y:number};
   p2: {x:number, y:number};
   p3: {x:number, y:number};
}

const area : (s : Shape) => number = (s : Shape) : number => {
   switch(s.tag) {
      case "circle": return s.radius * s.radius *  3.14;
      case "rectangle": return (s.upperLeft.x - s.lowerRight.x) * (s.upperLeft.y - s.lowerRight.y);
      case "triangle": return 0; // I do not know the formula :(
   }
}
```

מבחינת 'תורת הקבוצות', פעולות הבידול היא למעשה הפעולה disjoint union:

A **U+** B = (A X {0}) U (B X {1})

{0,1,2} **U** {2,3} = {0,1,2,3}

{0,1,2} **U+** {2,3} = {(0,0), (1,0), (2,0), (2,1), (3,1)}

<u>מבנים רקורסיביים</u>

ראינו כי בעזרת ה interface ניתן להגדיר טיפוסים רקורסיביים (Person, Student, Course)
נבחן דוגמא נוספת:

```
interface BinTree {
   root : number,
   left : BinTree,
   right : BinTree
}

let tree : BinTree = {
   root : 2,
   left : { root : 1 },
```

```
   right : { root : 3}
}
```

בעיה: העץ המושם לשדות left, right אינו שלם.
פתרון: נרחיב את הגדרת השדות left, right כך שהטיפוס יכלול גם null:

```
interface BinTree {
   root : number,
   left : BinTree | undefined,
   right : BinTree | undefined
}

let tree : BinTree = {
   root : 2,
   left : { root : 1, left : undefined, right : undefined },
   right : { root : 3, left : undefined, right : undefined }
}
```

לשם נוחות, קיים syntactic sugar עבור מקרים כאלו: האופציה **?**

```
interface BinTree {
   root : number,
   left? : BinTree,
   right? : BinTree
}
let tree : BinTree = {
   root : 2,
   left : { root : 1},
   right : { root : 3}
}
```

<u>טיפוסים גנריים</u>

ניתן להגדיר תבניות של טיפוסים עם משתנה המציין את אחד, או יותר, הטיפוסים במבנה:

```
interface BinTree<T> {
   root : T,
   left? : BinTree<T>,
   right? : BinTree<T>
}
```

באופן זה ניתן לאלץ את כל הערכים להיות מסוג אחד:

```
let tree : BinTree<number> = {
```

```
   root : 2,
   left : { root : 1},
   right : { root : 3}
};

let tree : BinTree<number> = {
   root : 2,
   left : { root : 1},
   right : { root : 'a'}   //incorrect
};

let tree : BinTree<number | string> = {
   root : 2,
   left : { root : 1},
   right : { root : 'a'}
};

let tree : BinTree<any> = {
   root : 2,
   left : { root : false},
   right : { root : 'a'}
};

   const square : (x:number)=>number = x : number => x*x;

   const squareTree : (t : BinTree<number>) => BinTree<number> = (t : BinTree<number>) =>
      (t.left === undefined && t.right === undefined) ?
         { root : square(t.root)}
      : (t.left === undefined) ?
         { root: square(t.root), right : squareTree(t.right) }
      : (t.right === undefined) ?
         { root : square(t.root), left : squareTree(t.left) }
      : {root: square(t.root), left : squareTree(t.left), right: squareTree(t.right)};
```

הערה: שימו לב כי העיצוב של הקוד הינו פונקציונאלי טהור – לא משתמשים בפעולת השמה על הערכים בקודקודים אלא מייצרים עץ חדש (כמו ה Immutable Objects בתכנות מערכות).
דוגמא נוספת לעיצוב שכזה – מימוש הפעולות push, pop על מחסנית, המחזירות מחסנית חדשה במקום לשנות את הקיימת. ראו את דוגמת הקוד ' Functional Stack: Step ' כאן.

<u>יחסי טיפוסים בין טיפוסים גנריים</u>:

BinTree<number> ------ instantiation -------- BinTree<T>
BinTree<string> -------- disjoint ------ BinTree<number>
BinTree<{ name : string , age : number}> ----- subtype ------> BinTree<{ name : string}>

BinTree<S> ------- subtype -----> BinTree<T>, S is a subtype of T

<u>Clusure</u>

נתבונן בדוגמת הקוד הבאה:

```
let z = 10,
   add = (x,y) => x+y+z;
  add(1,2)
```

המשתנה add הוא פונקציה המתייחסת בקוד שלה למשתנה z המאותחל ל 10. כך שהערך החוזר מהפעלתה עם 1,2 הוא 13.

כלומר, הפונקציה אינה מוגדרת רק ע"י הקוד שלה אלא גם ע"פ 'העולם' שהיה בזמן שהיא נוצרה (נכנה אותו בהמשך ה'סביבה' של הפרוצדורה, אוסף המשתנים עם הערכים שהיו קיימים בזמן שנוצרה, כמו z). הפונקציה היא 'סְגוֹר' closure של הגדרת הקוד שלה עם סביבה זו.

דוגמא נוספת:

```
let adder = (inc) => (x => x+inc),
   a5 = adder(5),
   a2 = adder(2);
```

a5 הוא closure המורכב מהפונקציה x => x+inc ומהסביבה {inc:5}

a2 הוא closure המורכב מהפונקציה x => x+inc ומהסביבה {inc:2}

כך שהקריאה a5(10) תחזיר 15, בעוד שהקריאה a2(10) תחזיר 12.

<u>תאימות טיפוסים</u>

- נאמר כי טיפוס T_2 ***תואם*** *(compatible)* לטיפוס T_1, אם ניתן להשים ערך מטיפוס T_2 למשתנה מטיפוס T_1. (דוגמאות להלן)
- תאימות טיפוסים אינה סימטרית. לדוגמא:

```
let s : Student = …
    p : Person = …;
    s = p    // incorrect, s is more specific
    p = s // correct, s is more specific
```

- חוקי התאימות

עקרון מנחה כללי: יחס ההכלה בין קבוצות הטיפוסים

- טיפוסים פשוטים תואמים רק לאותו טיפוס בדיוק

```
let x : number = 3,
    y : number = 4,
    s : string = 'a';
  x = y // correct
  x = s // incorrect
```

- טיפוסים פשוטים ומורכבים אינם תואמים

```
let x : number = 1,
    arr : number[] = [2];
    dict : { in : number } = { id :3 }
    x = arr // incorrect
```

```
x = dict // incorrect
```

- מערכים תואמים רק למערכים, מילונים למילונים, פונקציות לפונקציות

```
let arr : number[]  = [1,2],
    map : { a:number, b : number} = { a:1, b:2},
    f : (a : number) => number = (a) => a+1;

  arr = map // incorrect
  map = f // incorrect
```

  - מערך עם טיפוס יהיה תואם למערך אחר עם הטיפוס של אבריהם תואם

```
let pArr : Person[] = …,
  sArr : Student[] = …;
  pArr = sArr // correct
sArr = pArr // incorrect
```

  - מילון תואם למילון אם יש לו לפחות אותו מספר שדות, עם אותם שמות, ועם טיפוסים תואמים עבור ערכי השדות

```
let person : { id: number, name : string } = …,
    student : { id : number, name : string, univ : string } = …;
  person = student // correct
  student = person // incorrect

let room : { id : number, content : Person[] } = …,
    class : { id : number, content : Student[] } = …;
  room = class // correct
  class = room // incorrect
```

  - פונקציה תואמת לפונקציה כאשר:
    - מספר הפרמטרים זהה
    - הערך המוחזר תואם
    - הפרמטרים תואמים <u>באופן הפוך</u>

```
let f : () => Person = …,
   g : () => Student = …;
  f = g // correct
  g = f // incorrect

let f : (x : Person) => number = …,
   g : (x : Student) => number= …;
  f = g // incorrect
  g = f // correct
```

תופעה זו של היפוך התאימות של הפרמטרים מכונה Contravariance. הרציונאל הוא: הטיפוס של הפונקציה הוא קבוצת כל קטעי הקוד האפשריים העונים על דרישות 'החתימה'. כאשר

פונקציה מקבלת פרמטרים עשירים יותר (=יותר ספציפיים, עם יותר נתונים) יש יותר אפשרויות לכתוב את ה body שלה.

2. תחביר וסמנטיקה

2.1 מבוא

I

עד כה:

- דגמים שונים של שפות תכנות
- תחביר וסמנטיקה (ביטוי וערך, חישוב ביטוי לערך)
- טיפוסים (קבוצות, מערכת טיפוסים, תאימות טיפוסים)

בפרק זה:

- נגדיר שפת תכנות באופן שלם
  - הגדרת הרכיבים בשפה, התחביר
    - באופן לא פורמאלי
    - באופן פורמאלי
  - הגדרת משמעות הביטויים, הסמנטיקה (כיצד מחשבים ביטוי לערך)
    - באופן לא פורמאלי
    - באופן פורמאלי
  - מימוש פארסר ואינטרפרטר לשפה

Program → **Parser** (syntactic rules) → Abstract Syntax Tree
Abstract Syntax Tree → **Interpreter** (semantic rules) → Value

באופן זה נגדיר ארבע שפות: L1-L4

II רכיבי שפת התכנות

הרכיבים המרכזיים בשפת התכנות הם:

1. פרימיטיביים
   ביטויים מובנים בשפה, אבני הבסיס של התחביר.
   לדוגמא:
   ביטויים המתארים ערכים: מספרים, בוליאנים, מחרוזות...
   פונקציות: +, =, >, <,...
   [המשמעות של ביטויים אלו נקבעת על ידי האינטרפרטר]

2. אופני הרכבה
   כיצד לבנות ביטוי מורכב מביטויים פשוטים / ביטויים מורכבים קיימים ('תתי ביטויים).

3. אופני הפשטה
   כיצד ניתן לתאר ביטויים מורכבים כיחידה עצמאית / באופן פשוט יותר.

2.2 הגדרת שפות תכנות בסיסיות

2.2.1 השפה L1

I תחביר השפה L1

1. ביטויים פרימיטיביים

אטומיים:

Literal numbers: …,-3, -2, -1, 0, 1, 2, 3, 3.5, 2.7…
Literal boolean: #t, #f
Primitive procedures: +, -, *, /, >, <, =

מורכבים: מבנה הצורה define להלן

2. אופני הרכבה

בשפה L1 ניתן לבנות ביטויים מורכבים מביטויים פשוטים בעזרת סוגריים: ().
כאשר הביטוי השמאלי ביותר הוא פרוצדורה ושאר הביטויים בסוגריים הם הפרמטרים.
למעט מקרים של אופרטורים מיוחדים (להלן).

```
(+ 4 5)
➔ 9
(- 6 3)
→ 3
(* (/ 6 2) (+ 4 5))
→ 27
```

ג. אופני הפשטה
ב L1 ניתן לתת שם לביטוי, בעזרת ה*אופרטור המיוחד* define

```
(define pi 3.14)
(define n (* (/ 6 2) (+ 4 5)))
…
(* pi n)
→ 84.78
```

הקישור בין שם לבין ביטוי/ערך מכונה binding.

II סמנטיקת השפה L1

יש להגדיר עבור כל סוג של ביטוי אפשרי בשפה, כיצד הוא יחושב לערך.

1. חישוב ביטויים אטומיים לערך

- פרימיטיביים (ע"פ מה שימומש באינטרפרטר)
  - מספרים – הערך החשבוני

- בוליאני – הערך הלוגי (#t true, #f false)
- אופרטורים (נניח '+', דיון להלן)

- משתנים
לדוגמא: pi
ע"פ הערך שהוגדר עבורו ב define (כלומר lookup, להלן)

- אופרטור מיוחד (המילה 'define')
לא מחושבים - מופיעים בתוכנית אך ורק במסגרת מבנה מיוחד (אחרת הם יתפרשו כשם של משתנה), וגם אז רק לשם זיהוי סוג המבנה המיוחד.

2. חישוב ביטויים מורכבים לערך

- צורות מיוחדות (special form, סוגריים שבמקום השמאלי נמצא אופרטור מיוחד, כמו define) מחושבות כל אחת על פי משמעות הספציפית.
בשפה L1 יש רק צורה מיוחדת אחת: define
חישוב מבנה ה define מוגדר באופן הבא:
  - חישוב הביטוי מימין
  - הוספת הזוג <משתנה, ערך> ל"סביבה" (מבנה נתונים השומר את כל ה binding)

```
(define x (+ 1 2))
Eval (+ 1 2)
Add binding <x,3> to the environment
```

- הפעלה של פרוצדורה (ברירת המחדל)
  - חישוב האופרטור
  - חישוב כל אחד מהפרמטרים
  - הפעלת הפרוצדורה/אופרטור על הפרמטרים

```
(+ (* 2 3) pi)
→9.14
(= 2 (+ 1 1))
→ #t
```

<u>נקודות למחשבה</u>

- האם סדר החישוב של הביטויים בתוכנית משנה
עקרונית, סדר החישוב אינו משנה (אין פעולות השמה), למעט פעולת define המבצעת side-effect – הרחבת הסביבה.

```
(define pi 3.14)
…
(+ pi 5)
```

- האם הפרימיטיביים הם חלק מהשפה או עניינו הפרטי של האינטרפרטר

שאלה פילוסופית... (מקבילה לשאלה האם מילה חדשה בשפת בני אדם הופכת אותה לשפה חדשה)
בקורס נתייחס להוספת פרימיטיביים כעדכון של האינטרפרטר אך לא כיצירת שפה חדשה. מה שמגדיר את השפה ומבחין אותה משפות אחרות הם המבנים התחביריים השונים, ובפרט אוסף ה special forms. במילים אחרות, הוספה של special form חדש לשפה יוצרת שפה חדשה.

חסרונות השפה L1:

- יש רק פרוצדורות פרימיטיביות. לא ניתן להגדיר חדשות.
- סוגי ערכים מצומצמים (רק מספרים ובוליאני)
- אין 'מבני בקרה' כמו if, for
- לא ניתן לתחום את ה binding לקטע קוד מסוים (כל המשתנים גלובליים)
- באופן כללי ועקרוני: **השפה L1 אינה Turing-Complete (או בניסוח אחר, אינה lambda calculus)**

2.2.2 השפה L2

בשפה L2 נוסף מבנה בקרה (הצורה המיוחדת if) ואפשרות להגדרת פרוצדורות חדשות (הצורה המיוחדת lambda)

I תחביר

1. נגדיר מבנה תחבירי חדש, צורה מיוחדת, עבור הגדרת פרוצדורות
   (lambda (<var> …) <exp> …)

לדוגמא:

(lambda (x) (* x x))

כמו כל ביטוי אחר, ניתן לתת שם לביטוי של הגדרת פרוצדורה:

(define square (lambda (x) (* x x)))

כמו כל אופרטור פרימיטיבי, ניתן להפעיל את הפרוצדורות שאנחנו מגדירים:

```
(
 (lambda (x y) (* x y))
 3 4
)
→
12

(square 3 4)
→
12
```

2. נגדיר את הצורה המיוחדת if

```
(if <test> <then> <else>)

(if (> 3 1) #t #f)
(if (= (square 3) 9) (+ 1 2) (\ 3 0))
```

II סמנטיקה

1. אופן החישוב של lambda

- הגדרת פרוצדורה, לדוגמא: ((lambda (x) (* x x))
הערך של ביטוי זה הוא closure, הכולל בתוכו את: רשימת הפרמטרים, גוף הפרוצדורה, (ואת הסביבה הנוכחית - נדון בהמשך).
- הפעלת פרוצדורה, לדוגמא: ((lambda (x) (* x x)) 3)
  - החלפת/הצבת כל מופע של הפרמטר בגוף הפרוצדורה בערך שנשלח עבורו, לדוגמא: (* x x) ← (* 3 3)
  - חישוב הביטויים בגוף הפרוצדורה, לדוגמא: (* 3 3) ← 9
  - הערך החוזר הוא ערכו של הביטוי האחרון, לדוגמא: 9

2. אופן החישוב של if
- חישוב התנאי הבוליאני (test)
- אם הוא בעל ערך אמת: חישוב ביטוי ה then (שהוא ערך ביטוי ה if כולו)
- אחרת: חישוב ביטוי ה else (שהוא ערך ביטוי ה if כולו)

השפה 2L היא Turing-Complete!

<u>תכנות ב 2L</u>: דוגמא – פונקציה המחשבת קירוב של שורש של מספר נתון, ע"פ מתודת ניוטון:
- נתון מספר x
- ננחש את השורש y (שונה מ-0)
  - נקרב את y לשורש האמיתי של x על ידי: (y + x/y) / 2
  - וחוזר חלילה, עד שמספיק קרוב

```
(define sqrt (lambda (x) (sqrt-iter x 1)))

(define sqrt-iter
 (lambda (x root)
  (if (good? x root)
      root
    (sqrt-iter x (improve x root)))))

(define good?
  (lambda (x root)
    (< (abs (- (* root root) x) 0.0001))))
```

```
(define abs (lambda (x) (if (< x 0) (* -1 x) x)))

(define improve
  (lambda (x root)
    (average root (/ x root))))

(define average
  (lambda (x y)
    (/ (+ x y) 2)))
```

חסרון: התכנות ב 2L לא תמיד נוח

- אין לולאות (לא בהכרח חיסרון, ניתן לעצב כל קריאה רקורסיבית כרקורסיית זנב – פרק 4)
- לא ניתן לשנות ערך של משתנה (לא בהכרח חיסרון, שפה פונקציונאלית)
- מבנה נתונים מורכב (כמו רשימה) כחלק מהשפה
- משתנים לוקאליים

← בשפה 3L יתווספו <u>מבנה נתונים בסיסי</u> ו<u>משתנים לוקאליים</u>.

2.2.3 השפה 3L

בשפה זו נוסיף ביטויים עבור ערכים מורכבים, ומשתנים מקומיים.

1. ערכים מורכבים

ב JS היו שני סוגים של ערכים מורכבים: מערכים ומילונים.
ב 3L נגדיר מבנה בסיסי יותר: זוג ביטויים/ערכים

<u>תחביר</u>:

נגדיר מבנה נתונים של 'זוג' ביטויים/ערכים בעזרת האופרטורים הפרימיטיביים הבאים (כלומר, אנחנו מרחיבים את אוסף האופרטורים הבסיסי, שכלל +,-,*...):

cons – בנאי הבונה זוג
car – מקבל זוג ומחזיר את האיבר הראשון
cdr – מקבל זוג ומחזיר את האיבר השני
?pair – מקבל ביטוי ומחזיר #t אם הביטוי הוא זוג
?equal – מקבל שני זוגות מחזיר #t אם הם שווים

הערה, לשם נוחות, נגדיר גם literal expression עבור זוג: '(1 . 5), תיאור הזוג <1,5>

```
(define p1 '(1 . 5))
(define p2 (cons 2 6))
(car p1)
```

```
→ 1
(cdr p2)
→ 6
(define p3 (cons p1 p2))
(car p3)
→ '(1 . 5)
(cdr p3)
→ '(2 . 6)
(pair? 1)
→ #f
(pair? (cdr p3))
→#t
(equal? (car p3) (cdr p3))
→ #f
(equal? (cons 1 2) '(1 . 2))
→ #t
```

<u>הערה</u>: ניתן היה בפשטות לממש את האופרטור הפרימיטיבי החדש ?equal כפרצודרת משתמש:

```
(define equal?
  (lambda (p1 p2)
    (or (= p1 p2)
        (and (pair? p1)
             (pair? p2)
             (equal? (car p1) (car p2))
             (equal? (cdr p1) (cdr p2))))))
```

אך נוח יותר להגדיר פעולה זו כאופרטור פרימיטיבי, בפרט לשם תמיכה בהמשך של בדיקת השוויון של אובייקטים מורכבים נוספים.

בעזרת זוגות, ניתן לתאר בנוחות יחסית כל מבנה נתונים.

לשם נוחות יתר, נרחיב את השפה כך שתתמוך ב<u>רשימות</u>.

הגדרה אינדוקטיבית של רשימה:

- רשימה יכולה להיות <u>הרשימה הריקה</u>
- או <u>זוג</u> של <u>ביטוי</u> ו<u>רשימה</u>

הרשימה 1,2,3 לדוגמא תהיה:

```
(cons 1 (cons 2 (cons 3 '())))
```

🡸 נוסיף לשפה פרימיטיב חדש עבור הרשימה הריקה:

Literal-expression: '()

אופרטור: ?empty

עם שני פרימיטיבים אלו (literal-expression לרשימה ריקה, והשאילתא) ניתן להגדיר כעת רשימה, ע"פ ההגדרה לעיל:

```
(define list?
  (lambda (lst)
     (or (empty? lst)
         (and (pair? lst)
               (list? (cdr lst))))))

;; constructor for list with given two items
(define list
  (lambda (x y)
   (cons x (cons y '()))))
```

לשם נוחות, נגדיר גם את list, list? כאופרטורים פרימיטיביים (זה יאפשר בין היתר לקבל בבנאי list מספר כלשהו של איברים).
נרחיב את מגוון השאילתות, ע"פ פרוצדורות שלנו:

```
(define head car)
(define tail cdr)
(define first (lambda (l) (car l))
(define second (lambda (l) (car (cdr l)))
…
```

וכן פעולות שונות:

```
(define length
  (lambda (lst)
   (if (empty? lst)
      0
      (+ 1 (length (cdr lst))))))

(define nth
  (lambda (lst n)
   (if (empty? lst)
      '()
      (if (= n 0)
         (car lst)
         (nth (- n 1) (cdr lst))))))
```

2. משתנים לוקאליים

מוטיבציה

- ביטוי החוזר מספר פעמים

```
(define f
  (lambda (x y)
```

```
(+ (* x
     (square (+ 1 (* x y))))
    (* y
      (- 1 y))
    (* (+ 1 (* x y))
       (- 1 y)))))
```

← (i) חזרה מיותרת על קוד קיים (פחות קריא, שגיאות בהעתקה, יותר טסטים); (ii) חישוב כפול מיותר בזמן ריצה.

- פתרון א: הגדרתו כפרוצדורה

```
(define f1
  (lambda (x y) (+ 1 (* x y))))

(define f2
  (lambda (y) (- 1 y)))

(define f
  (lambda (x y)
    (+ (* x (square (f1 x y)))
       (* y (f2 y))
       (* (f1 x y) (f2 y)))))
```

- יתרון: אין צורך לכתוב מחדש כל פעם את הביטויים, הם מוגדרים פעם אחת בפונקציה
- חיסרון: חישוב חוזר של הקריאה לפרצודורה (כל פעם שהביטוי מופיע)
- פתרון: הגדת הביטוי כמשתנה לוקאלי.

← הצורה המיוחדת **let**, המאפשרת להגדיר משתנים לוקאליים.

תחביר:

```
(let ( (<var1> <exp1>)
       (<var2> <exp2>)
       ...
       (<varn> <expn>) )
  <body>)
```

מבנה ה let כולל שני חלקים: הגדרת המשתנים הלוקליים (bindings) וגוף הקוד (body)

```
(define f
  (lambda (x y)
    (let
      (
       (v1 (+ 1 (* x y)))
       (v2 (- 1 y))
      )
    (+ (* x
         (square v1)))
       (* y v2)
```

```
    (* v1 v2))))
```

סמנטיקה: מה הערך של הצורה המיוחדת let

- חישוב הערכים ב bindings (ע"פ הסביבה הנוכחית)
- הגדרת המשתנים ב binding כך שהם מקושרים לערכים לחושבו
- הצבת הערכים של המשתנים הלוקאליים ב body של ה let, וחישוב של ה bodyלאחר ההצבה.
- הערך של כל מבנה ה let הוא הערך של הביטוי האחרון ב body

<u>אבחנה</u>: מבנה ה let הוא סה"כ קיצור-תחבירי (syntactic abbreviation), כלומר ניתן היה להגדיר אותו עם הפרימיטיביים הקיימים כבר בשפה. ובפרט, כהפעלה של פונקציה:

```
(define f
  (lambda (x y)
    (
     (lambda (v1 v2)
      (+ (* x
           (square v1)))
      (* y
        v2)
      (* v1 v2))
     (+ 1 (* x y)) (- 1 y)
    )
  )
)
```

<u>לסיום</u>, לשם נוחות, נרחיב במקצת את אוסף סוגי הערכים בשפה 3L:

עד כה היו שני סוגי ערכים ב-L2: מספרים ובוליאנים.
נוסיף ב-L3 גם <u>מחרוזות</u> ו<u>סמלים</u>.

```
String: “ab c”
Symbol: 'green

(define name “Yossi”)
(define color ‘green)

(define f
  (lambda (color)
    (if (= color ‘green)
        (+ 3 5)
        (if (= color ‘red)
```

```
(* 7 9)
0))))
```

2.2.3.1 פונקציות מסדר גבוה (High Order Functions) ב 3L

כזכור, פונקציות מסדר גבוה מקבלות פונקציות כפרמטר ו/או מחזירות פונקציות כערך. באופן זה, ניתן לבנות תשתית גבוהה/מופשטת של פונקציות מעל תשתית נמוכה יותר.
נבחן כמה פונקציות קלאסיות שכאלה:

1. map

```
;;Signature: map(f,l)
;;Type: [ (T1->T2) * List(T1) -> List(T2) ]
;; Purpose: Apply f to all elements in lst. Return the list of results
;; Pre-condition: none
;; Tests: (map square '(1 2 3)) → '(1 4 9); (map (lambda (x) (> x 0)) '(-3 4)) → '(#f #t)
(define map
  (lambda (f lst)
    (if (empty? lst)
        '()
        (cons (f (car lst))
              (map f (cdr lst))))))
```

דוגמת ריצה:

```
(map (lambda (x) (* x x)) '(2 3))

lst = '(2 3)
f = (lambda (x) (* x x))

(cons 4 (map f '(3)))

lst = '(3)
f = (lambda (X) (* x x))

(cons 9 (map f '()))

lst = '()
f = (lambda (x) (* x x))

'()

→ (cons 4 (cons 9 '()))
→ '(4 9)
```

ב. filter

```
;;Signature: filter(pred,lst)
```

```
;;Type: [(T1->Boolean)*List(T1) -> List(T1) ]
;;Purpose: Return the elements of l which satisfy the predicate pred
;; Pre-condition: none
;; Tests: (filter (lambda (x) (> x 0)) ‘(-2 4)) → ‘(4)
(define filter
  (lambda (pred lst)
    (if (empty? lst)
        ‘()
        (if (pred (car lst))
            (cons (car lst) (filter pred (cdr lst)))
            (filter pred (cdr lst))))))

(filter (lambda (x) (> x 0)) ‘(1 -1))
-> ‘(1)
```

reduce .3

```
;;Signature: reduce(reducer,init,lst)
;;Type: [(T1*T2->T2)*T2*List(T1)-> T2]
;;Purpose: Combine the values of l, starting with init, according to the reducer
;; Pre-condition: none
;; Tests: (reduce + 0 ‘(1 2 3)) → 6; (reduce (lambda (x b) (and b (> x 0)) #t ‘(1 -2)) → #f
(define reduce
  (lambda (reducer init lst)
    (if (empty? lst)
        init
        (reducer (car lst)
                 (reduce reducer init (cdr lst))))))

(reduce + 0 '(1 2))

reducer: +
init: 0
lst: '(1 2)

→(+ 1 (reduce + 0 '(2)))

reducer: +
init: 0
lst: '(2)

→(+ 2 (reduce + 0 '()))

reducer: +
```

init: 0
l: '()

→ 0

→ 3
[1 + 2 + 0]

```
;;Signature: reduce2(reducer,init,l)
;;Type: [(T1*T2->T2)*T2*List(T1)-> T2]
;;Purpose: Combine the values of l, starting with init, according to the reducer
;; Pre-condition: none
;; Tests: (reduce2 + 0 '(1 2 3)) → 6; (reduce (lambda (x b) (and b (> x 0)) #t '(1 -2)) → #f
(define reduce2
  (lambda (reducer init lst)
    (if (empty? lst)
        init
        (reduce2 reducer
                 (reducer (car lst) init)
                 (cdr lst)))))
```

(reduce2 + 0 '(1 2))

reducer: +
init: 0
lst: '(1 2)

→(reduce2 + 1 '(2))

reducer: +
init: 1
lst: '(2)

→(reduce2 + 3 '())

→3
[0 + 1 + 2]

האם reduce, reduce2 שקולות?
ראינו כי סדר הפעלת פונקציית המיזוג (reducer) על איברי הרשימה שונה בשני המימושים.
← אם היא אינה אסוציטיבית/קומוטטיבית (כמו לדוגמא חלוקה) הפונקציות אינן שקולות.
(reduce / 1 '(2 3)) → 2 \ (3 \ 1)
(reduce2 / 1 '(2 3)) → 3 \ (2 \ 1)

4. הכללת דפוס פעולה

נתבונן בפונקציות הבאות:

הפונקציה sum-integers מקבל תחום של מספרים [a,b] וסוכמת את המספרים בתחום זה:

```
;; Signature: sum-integers(a,b)
;; Type: (Number*Number)->Number
;; Purpose: compute the sum of all integers a to b
(define sum-integers
  (lambda (a b)
    (if (> a b)
        0
        (+ a (sum-integers (+ a 1) b)))))
```

הפונקציה sum-cubes מקבל תחום של מספרים [a,b] וסוכמת את הערך המעוקב של המספרים בתחום זה:

```
(define cube (lambda (x) (* x x x)))

;; Signature: sum-cubes(a,b)
;; Type: (Number*Number)->Number
;; Purpose: compute the sum of the cube of all integers a to b
(define sum-cubes
  (lambda (a b)
    (if (> a b)
        0
        (+ (cube a) (sum-cubes (+ a 1) b)))))
```

הפונקציה pi-sum מחשבת קירוב למספר pi ע"פ הטור הבא:

1/a(a+2) + 1/(a+4)(a+6) + 1/(a+8)(a+10)…

מתכנס ל pi/8 כאשר מתחילים מ a=1

```
;; Signature: pi-sum(a,b)
;; Type: (Number*Number)->Number
;; Purpose: compute the sum of 1/a*(a+2) + ... + 1/(a+4n)*(a+4n+2)
;;        s.t. a+4n <= b < a + 4(n + 1)
(define pi-sum
  (lambda (a b)
    (if (> a b)
        0
        (+ (/ 1 (* a (+ a 2)))
           (pi-sum (+ a 4) b)))))
```

שלושת הפונקציות הנ"ל בנויות באותה תבנית:

```
(define <name>
  (lambda (a b)
    (if (> a b)
        0
        (+ (<f> a)
           (<name> (<next> a) b))))
```

← נגדיר פונקציה כללית מסדר גבוה, המקבלת את next ו f (כיצד להתקדם, ומה מבצעים על כל איבר) כפרמטרים:

```
;; Signature: sum(f, a, next, b)
;; Type: [ [Number->Number] * Number * [Number->Number] * Number -> Number]
;; Purpose: Compute the sum: (term a) + (term (next a)) + .... + (term n)
;;        where n = (next (next (... (next a)))) <= b and (next n) > b.
(define sum
  (lambda (f a next b)
    (if (> a b)
        0
        (+ (f a)
           (sum f (next a) next b)))))


(define sum-integers
  (lambda (a b)
    (sum (lambda (x) x) a (lambda (n) (+ n 1)) b)))

(define sum-cubes
  (lambda (a b)
    (sum cube a (lambda (n) (+ n 1)) b)))

(define pi-sum
  (lambda (a b)
    (sum (lambda (x) (/ 1 (* x (+ x 2)))) a (lambda (n) (+ n 4)) b)))
```

נשתמש בתבנית זו עבור קירוב לחישוב האינטגרל של פונקציה נתונה f בתחום [a,b]:

[f(a+dx/2) + f(a+dx+dx/2) + f(a+2dx+dx/2)… + f(a+ndx+dx/2)]dx

כאשר התחום [a,b] מחולק ל-n יחידות של dx

כדי להשתמש בתבנית sum שהגדרנו, נגדיר את f,next,a,b באופן הבא:

'f': f
'next': +dx
'a': a+dx/2
'b': a + ndx+ dx/2 = b

```
;; Signature: integral(f,a,b)
;; Type: [ [Number->Number] * Number * Number -> Number]
;; Purpose: Compute an approximation of the definite integral of f between a and b.
(define integral
  (lambda (f a b dx)
    (* (sum f (+ a (/ dx 2)) (lambda (x) (+ x dx) b)
        dx)))
```

דוגמא אחרונה: חישוב קירוב לנגזרת

הפונקציה derive מקבלת פונקציה f, דרגת קירוב dx, ומחזירה קירוב לפונקציית הנגזרת, ע"פ הנוסחה הבאה:

$f'(x)=[f(x+dx)-f(x)]/dx$

```
;; Signature: derive(f, dx)
;; Type: [ (Number->Number) * Number -> (Number->Number) ]
;; Purpose: Construct a function that computes a numerical approx of the derivative of f with
;; resolution dx.
(define derive
  (lambda (f dx)
    (lambda (x)
      (/ (- (f (+ x dx))
            (f x))
         dx))))
```

דוגמת הפעלה:

```
(define d-square-001 (derive square 0.001))
(define d-square-1 (derive square 0.1))

(d-square-001 2)
→ 4.000999999999699 (Real value is 4)
(d-square-1  2)
→ 4.100000000000001
```

הפונקציה nth-deriv מקבלת פונקציה ומחזירה את הנגזרת ה-nית שלה.

בגרסה הראשונה, מוחזרת פונקציה, או קוד, המבצע את כל תהליך ה-n גזירות בכל פעם שנדרש להפעיל את פונציית הנגזרת ה-nית על X נתון.

```
;; Signature: nth-deriv(f,dx,n)
;; Type: [ (Number->Number) * Number * Number -> (Number->Number) ]
;; Purpose: construct a function that computes a numerical approximation of the nth derivative
of f, according to approximation dx
(define nth-deriv
  (lambda (f n dx)
    (lambda (x)
      (if (= n 0)
          (f x)
        ( (nth-deriv (derive f dx) (- n 1) dx) x)
      ) ;; if
     ) ;; lambda
    ) ;; lambda
  ) ;; define
```

בגרסה השניה, היעילה יותר, מוחזרת כבר פונקציית הנגזרת ה-nית, כך שבהינתן x בהמשך, תופעל עליו מיד פונקציית הנגזרת ה-nית:

```
(define nth-deriv-early
  (lambda (f n dx)
    (if (= n 0)
        f
        (derive (nth-deriv-early f (- n 1) dx) dx))))
```

2.3 הגדרה פורמאלית של תחביר, ומימושו

2.3.1 מבוא

<u>התחביר</u>:

1. מגדיר את ה'מילים'/ה'טוקנים' החוקיות בשפה

דוגמאות:

- שפה אנושית (עברית, ערבית, רוסית, אנגלית): המילון, אוצר המילים בשפה, כולל
  הטיות
  כסא
  שולחן
  הכסא
  שולחני
  שולחנות

...

- פרוטוקול תקשורת בין תהליכים
  לדוגמא, הפרוטוקול מתרגיל 3 ב SPL

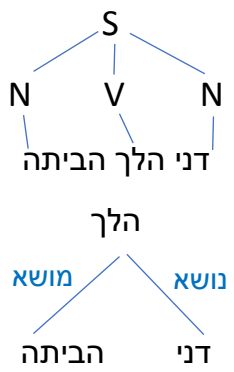
  login, register, ‘abc’, \n, \0….

- השפה 2L

  (,), define, lambda, if, 3, #f,…

2. מגדיר את מבנה ה'משפטים' בשפה, כלומר את הסדרים החוקיים של המילים, ואת התפקיד של כל חלק.

- שפה אנושית



- פרוקטוקול תקשורת

  login [command] danny [parameter]\n
  register [command] spl [parameter] ‘…’\n

- L2

  (if (> x 3) 4 5)
  [if-expr: test, then, else]

← הגדרת תחביר באופן פורמאלי, מבוסס על שני סוגים של חוקים:

1. חוקים לקסיקאליים
   כיצד לחלק את רצף התווים בתוכנית ל"טוקנים'
   - תווים מפרידים, משתנים, מספרים
   - ניתן לתיאור ע"י ביטוי רגולרי

2. חוקים תחביריים
   הגדרת המבנה ההיררכי של הטוקנים, והתפקיד של כל רכיב
   - ניתן לתיאור ע"י דקדוק חסר הקשר

🡸המערכת שנבנה עבור ניתוח תוכנית נתונה והרצתה, תורכב מהרכיבים/השלבים הבאים:

- Scanner

מקבל מחרוזת של תוכנית/ביטוי ומחזיר מערך של טוקנים, ע"פ החוקים הלקסיקאליים.

"(if (> x 3) 4 5)"

→

['(', 'if', '(', '>', 'x', '3', ')','4','5', ')']

- Reader

מקבל מערך של טוקנים, ומחזיר את המבנה ההיררכי שלהם, ע"פ החוקים התחביריים.

['(', 'if', '(', '>', 'x', '3', ')','4','5', ')']

→

['if', ['>', 'x', '3'],'4','5']

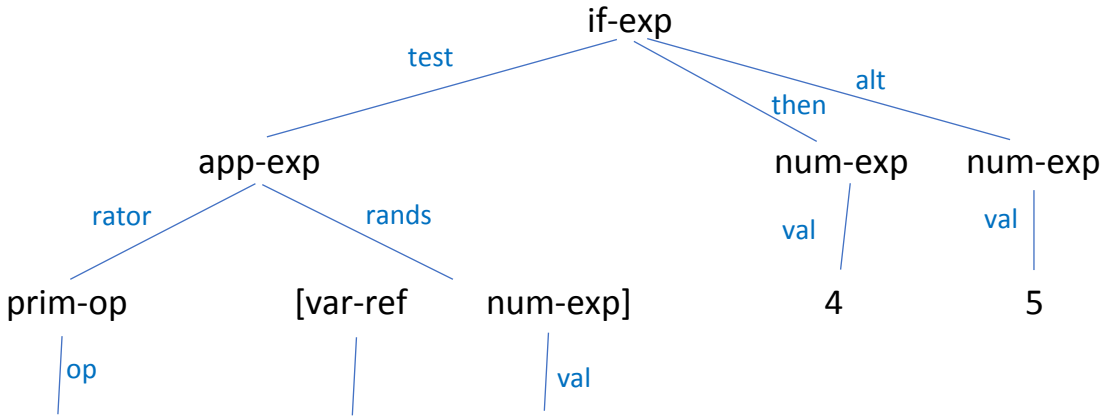
מבנה היררכי זה מכונה Symbolic Expression או בקיצור S-Exp

- Parser

מקבל s-exp ומחזיר עץ תחבירי מופשט (Abstract Syntax Tree, AST), הכולל מידע על התפקיד של כל רכיב, ע"פ החוקים התחביריים.

['if', ['>', 'x', '3'],'4','5']

→



< x 3

○

Interpreter

מקבל AST ומחזיר ערך, ע"פ החוקים הסמנטיים

לדוגמא: עבור ה AST הנ"ל, בהנחה ש x הוגדר קודם כ 6 (בעזרת define) – האינטרפרטר יחזיר את הערך 5.

2.3.2 הגדרה פורמאלית של חוקי התחביר

1. חוקים לקסיקאליים

ניתן ע"י ביטוי רגולרי.
לשם נוחות, נשתמש בדקדוק חסר הקשר:

```
<token> ==> <identifier> | <boolean> | <number> | <string> |( | ) | ' | .
<delimiter> ==> <whitespace> | ( | ) | "
<whitespace> ==> <space or newline>
<identifier> ==> <initial> <subsequent>*
<initial> ==> <letter>
<letter> ==> a | b | c | ... | z
<subsequent> ==> <initial> | <digit>
<digit> ==> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<string> ==> " <string element>* "
<string element> ==> <any character other than " or \> | \" | \\
<boolean> ==> #t | #f
```

2. חוקים תחביריים

על ידי דקדוק חסר הקשר.
נאמץ את הפורמט של בנק-נאור: Bank-Naor Form (BNF)

קיימות שתי דרכים להגדיר את חוקי התחביר:

מבנה תחבירי קונקרטי:
- תלוי שפה (משתמש בביטויים בשפה, כמו keywords, סימני פיסוק...)
- נוח יותר לקריאה והבנה
- מיועד למשתמשים בשפה, כלומר לכותבי הקוד

מבנה תחבירי מופשט:
- התיאור אינו תלוי שפה
- התיאור מבוסס על מבני נתונים
- מקצועי, מיועד לאנשי המטה-פרוגרמינג (כמו כותבי הפארסר והאינטרפרטר)

```
<exp> ::= <define> | <cexp>              / def-exp | cexp
<define> ::= ( define <var> <cexp> )     / def-exp(var:varDecl, val:cexp)
<var> ::= <identifier>                   / varRef(var:string)
<binding> ::= ( <var> <cexp> )           / binding(var:varDecl, val:cexp)
<cexp> ::= <number>                      / num-exp(val:number)
  |  <boolean>                           / bool-exp(val:boolean)
  |  <string>                            / str-exp(val:string)
  |  <varRef>                            / varRef(var)
  |  ( lambda ( <varDecl>* ) <cexp>+ )   / proc-exp(params:List(varDecl), body:List(cexp))
  |  ( if <cexp> <cexp> <cexp> )         / if-exp(test: cexp, then: cexp, else: cexp)
  |  ( let ( binding* ) <cexp>+ )        / let-exp(bindings:List(binding), body:List(cexp))
  |  ( <cexp> <cexp>* )                  / app-exp(operator:cexp, operands:List(cexp))
  |  ( quote <sexp> )                    / literal-exp(val:sexp)
```

2.6.3 מימוש ה Scanner, Reader, Parser

1. Scanner, Reader

מאחר והשפות שלנו (L1-L3) הינן 'שפות סוגריים', כלומר שפות שבהן ההיררכיה של הטוקנים נקבעת אך ורק ע"י סוגריים, קל מאוד להמיר את התוכנית למבנה של S-Exp.
לשמחתנו קיים כבר כלי שכזה ב TypeScript.

2. Parser

מימוש הפארסר ממוקד בקובץ L3-ast.ts

מבוא לקוד:

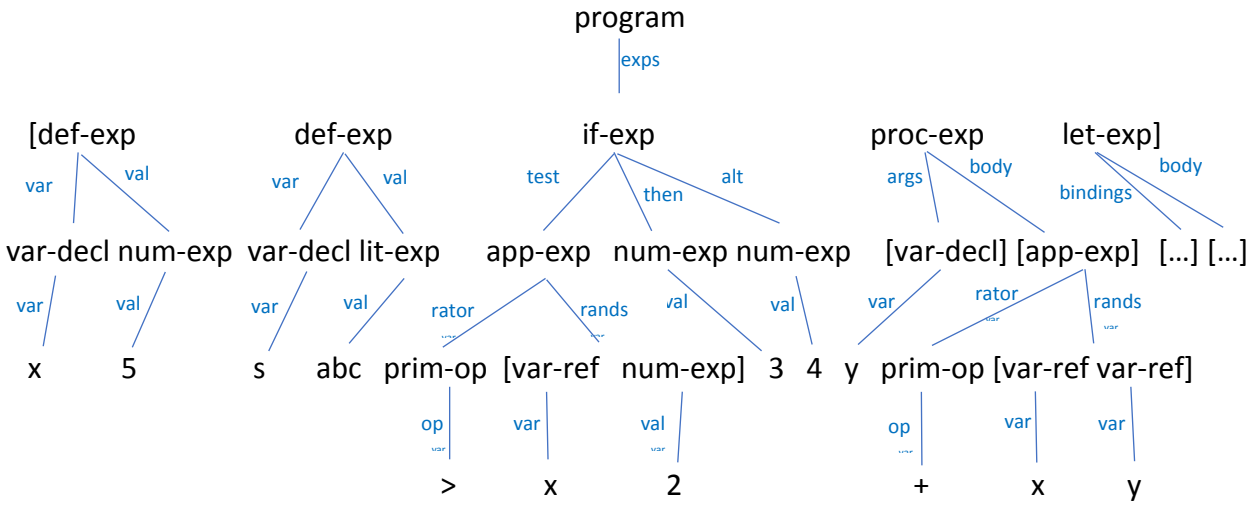
- תשתית
  - מבנה נתונים עבור כל סוג ביטוי בתחביר המופשט
    - בנאי
    - שאילתת טיפוס
  - הכללת מבנים שונים ע"י disjoint union (כלומר, הוספת שדה tag לכל interface, וביצוע איחוד של טיפוסים)
  - טיפול בשגיאות (בעזרת המבנה Result ואוסף פונקציות המטפלות בתרחיש בו המידע הנתון הינו או תוצאה או שגיאה)

- הרעיון הכללי
  - בכל שלב יש לעבד s-exp נתון.
  - ה s-exp יכול להיות מחרוזת או מערך
    - מחרוזת: הגדרת AtomicExp בהתאם לתוכן המחרוזת (NumExp, BoolExp, PrimOp, VarRef, StrExp)
    - מערך: בניית CompoundExp על פי האיבר הראשון במערך - צורה מיוחדת (IfExp, ProcExp, LetExp, LitExp), או AppExp כברירת מחדל – תוך קריאה רקורסיבית לביצוע הפארס על רכיבי המבנה (שאר ה s-exp במערך הנתון)

הדגמת הקוד:

```
"(L3
   (define x 5)
   (define s ‘abc)
   (if (> x 2) 3 4)
   (lambda (y) (+ x y))
   (let ((z 2) (y 5))
      (* x y z))”
```

```
[‘L3’, [‘define’, ‘x’, ‘5’],
        [‘define’, ‘s’, [‘quote’, ‘abc’]],
        [‘if’, [‘>’, ‘x’, ‘2’], ‘3’, ‘4’],
        [‘lambda’, [‘y’], [‘+’, ‘x’, ‘y’]],
        [‘let’, [[‘z’, ‘2’], [‘y’, ‘5’]], [‘*’, ‘x’, ‘y’]]
]
```



2.3.3 ביצוע טרנספורמציות תחביריות על AST נתון של תוכנית

ה-AST מייצג באופן נוח את התוכנית שאותה אנו מעוניינים לעבד.
ה interpreter ייקח AST ויהפוך אותו לערך.
היום נבחן כמה פעולות אחרות שניתן לבצע על AST של תוכנית.

1. חישוב 'גובה' ה AST (כלומר 'עומק' התוכנית)

```
export const height = (exp: Program | Exp): number =>
  isAtomicExp(exp) ? 1 :
  isLitExp(exp) ? 1 :
  isDefineExp(exp) ? 1 + height(exp.val) :
  isIfExp(exp) ? 1 + Math.max(height(exp.test), height(exp.then), height(exp.alt)) :
  isProcExp(exp) ? 1 + reduce(Math.max, zero,
                    map((bodyExp) => height(bodyExp), exp.body)) :
  isLetExp(exp) ? 1 + Math.max(
                 reduce(Math.max, zero,
                      map((binding) => height(binding.val), exp.bindings)),
                 reduce(Math.max, zero,
                      map((bodyExp) => height(bodyExp), exp.body))) :
  isAppExp(exp) ? 1+ Math.max(height(exp.rator),
                  reduce(Math.max, zero,
                       map((rand) => height(rand), exp.rands))) :
  isProgram(exp) ? 1 + reduce(Math.max, zero,
                    map((e) => height(e), exp.exps)) :
  -1;
```

2. בדיקה האם משתנה נתון 'מופיע חופשי' (occurs free) בביטוי נתון

<u>הגדרות</u>:

משתנה x <u>מופיע חופשי</u> (occurs free) בביטוי E אם ורק אם:

- יש התייחסות (VarRef) ל-x בביטוי E
- x אינו מוגדר (VarDecl) בביטוי [הגדרת ביטוי אפשרית ב L3 בשלושה מבנים: define, lambda, let]

משתנה x <u>מופיע כבול</u> (occurs bound) בביטוי E אם ורק אם:

- יש התייחסות (VarRef) ל-x בביטוי E
- x מוגדר (VarDecl) בביטוי

דוגמאות:

```
((lambda (x) x) y)
```

- `x` occurs bound - since the 2nd occurrence of x in the body of the lambda is bound by the first occurrence in the formals of the lambda.
- `y` occurs free.

```
(lambda (y)

  ((lambda (x) x) y))
```

- The reference of `y` in the second line is now bound by the declaration of `y` in the first line.

```
export const occursFree = (v: string, e: Program | Exp): boolean =>
   isBoolExp(e) ? false :
   isNumExp(e) ? false :
   isStrExp(e) ? false :
   isPrimOp(e) ? false :
   isLitExp(e) ? false :
   isVarRef(e) ? (v === e.var) :
   isIfExp(e) ? occursFree(v, e.test) || occursFree(v, e.then) || occursFree(v, e.alt) :
   isProcExp(e) ? !includes(v, map((p) => p.var, e.args)) &&
                 some((b) => occursFree(v, b), e.body) :
   isAppExp(e) ? occursFree(v, e.rator) ||
                some((rand) => occursFree(v, rand), e.rands) :
   isDefineExp(e) ? (v !== e.var.var) && occursFree(v, e.val) :
   isProgram(e) ? some((b) => occursFree(v, b), e.exps)  :
   isLetExp(e) ? !includes(v,map((binding) => binding.var.var, e.bindings)) &&
                     some((b) => occursFree(v, b), e.body) :
   false;
```

3. מציאת כל המשתנים שיש אליהם ref בביטוי נתון

```
export const referencedVars = (e: Program | Exp) : VarRef[] =>
   isBoolExp€ ? Array<VarRef>() :
   isNumExp€ ? Array<VarRef>() :
   isStrExp€ ? Array<VarRef>() :
   isLitExp€ ? Array<VarRef>() :
   isPrimOp€ ? Array<VarRef>() :
   isVarRef€ ? [e] :
   isIfExp€ ? reduce(varRefUnion, Array<VarRef>(),
                  map(referencedVars, [e.test, e.then, e.alt])) :
   isAppExp€ ? union(referencedVars(e.rator),
                  reduce(varRefUnion, Array<VarRef>(), map(referencedVars, e.rands))) :
   isProcExp€ ? reduce(varRefUnion, Array<VarRef>(), map(referencedVars, e.body)) :
   isDefineExp€ ? referencedVars(e.val) :
   isProgram€ ? reduce(varRefUnion, Array<VarRef>(), map(referencedVars, e.exps)) :
   isLetExp€ ? union(
      reduce(varRefUnion, Array<VarRef>(),
                 map(referencedVars, map((binding) => binding.val, e.bindings))),
      reduce(varRefUnion, Array<VarRef>(), map(referencedVars, e.body))) :
    e;
```

4. המרת מבני ה let בתוכנית ל app-exp שקול

תזכורת: מבנה ה let הוא syntactic abbreviation למבנה של הפעלת פרוצדורות

(let

```
  ( (a 3) (b 4) )
     (+ a b)
)
→
(
 (lambda (a b)
    (+ a b)
 )
 3 4
)

/*
Purpose: rewrite a single LetExp as a lambda-application form
Signature: rewriteLet(cexp)
Type: [LetExp => AppExp]
*/
const rewriteLet = (e: LetExp): AppExp => {
   const vars : VarDecl[] = map((b) => b.var, e.bindings);
   const vals : Cexp[] = map((b) => b.val, e.bindings);
   return makeAppExp(
          makeProcExp(vars, e.body),
          vals);
}

/*
Purpose: rewrite all occurrences of let in an expression to lambda-applications.
Signature: rewriteAllLet(exp)
Type: [Program | Exp -> Program | Exp]
*/
export const rewriteAllLet = (exp: Program | Exp): Program | Exp =>
   isExp(exp) ? rewriteAllLetExp(exp) :
   isProgram(exp) ? makeProgram(map(rewriteAllLetExp, exp.exps)) :
   exp;

const rewriteAllLetExp = (exp: Exp): Exp =>
   isCExp(exp) ? rewriteAllLetCExp(exp) :
   isDefineExp(exp) ? makeDefineExp(exp.var, rewriteAllLetCExp(exp.val)) :
   exp;

const rewriteAllLetCExp = (exp: Cexp): Cexp =>
   isAtomicExp(exp) ? exp :
   isLitExp(exp) ? exp :
   isIfExp(exp) ? makeIfExp(rewriteAllLetCExp(exp.test),
                    rewriteAllLetCExp(exp.then),
```

```
                    rewriteAllLetCExp(exp.alt)) :
isAppExp(exp) ? makeAppExp(rewriteAllLetCExp(exp.rator),
                    map(rewriteAllLetCExp, exp.rands)) :
isProcExp(exp) ? makeProcExp(exp.args, map(rewriteAllLetCExp, exp.body)) :
isLetExp(exp) ? rewriteAllLetCExp(rewriteLet(exp)) :
exp;
```

הערה: נדרש לקרוא ל **rewriteAllLetCExp** לאחר המרת ה LextExp, כי ייתכן שהוא כולל בתוכו תת ביטוי נוסף מסוג LetExp. לדוגמא:

```
(let ( (a 1)
      (b (let ((c 3)) c))
     )
     ( + a b))
→
(
 (lambda (a b)
   (+ a b)
 )
 1 (let ((c 3)) c)
)
```

5. הוספת 'כתובת לקסיקאלית' לכל התייחסות למשתנה בתוכנית (מעין מצביע למקום שבו הוא מוגדר)

נגדיר 'כתובת לקסיקאלית' של VarRef כ'הצבעה' למקום בו הוא מוגדר:

- כמה רמות של הגדרה צריך לעלות כדי להגיע להגדרה שלו
- היכן הוא ממקום ברשימת ההגדרות ברמה זו

```
(lambda (x y)
  (
   (lambda (x) (+ x y)(+ x z))
   1
  )
)

(lambda (x y)
  (
   (lambda (x)
    ([+ free] [x : 0 0] [y : 1 1]))([+ free] [x : 0 0] [z :
free]))
  1
 )
)
```

מימוש:

- הרחבת מבני הנתונים בדקדוק עם מבנים התומכים בכתובות ה VarRef
  - FreeVar
  - LexicalAddress
  - CExpLA (הרחבה של Cexp כך שמתאפשר גם VarRef מסוג FreeVar ו LexicalAddress)

- מבנה נתונים
במהלך ביצוע הטרנספורמציה, כלומר הוספת הכתובות של המשתנים בכל VarDecl, מתוחזק מאגר הממפה את המשתנים בהם נתקלנו עד כה (תוך כדי המעבר על ה AST) למקום שבהם הם מוגדרים (כלומר ל Lexical Address שלהם). במילים אחרות, כל פעם שאנו מגיעים ל VarDecl (define, lambda, let) אנו מוסיפים את המשתנה וכתובתו למאגר, ומצד שני, כל פעם שאנו נכנסים לרמה עמוקה יותר (נניח ל lambda פנימית) יש לעדכן את הרמה של כל המשתנים במאגר.

- פונקציות עזר

crossContour

מבוצעת כאשר נכנסים לרמה חדשה של הגדרת משתנים (כל פעם שאנחנו מגיעים ל VarDecl):

- הוספת המשתנים החדשים למאגר, ברמה 0
- העמקת הרמה עבור המשתנים שהוגדרו קודם

```
crossContour(
   [[VarDecl a], [VarDecl b]],
   [[LexAddr a 0 0], [LexAddr c 0 1]]) =>

[[LexAddr a 0 0], [LexAddr b 0 1], [LexAddr a 1 0], [LexAddr c 1 1]]
```

getLexicalAddress

מופעלת בכל פעם שאנו מגיעים ל VarRef ונדרשים להמירו ל Lexical Address:
בהינתן מאגר של הגדרת הכתובות של המשתנים השונים, ושם של משתנה, מחזירה את הכתובת האחרונה ביותר של משתנה זה:

```
getLexicalAddress(<var-ref a>, [[lex-addr a 0 0], [lex-addr b 0 1], [lex-add a 1 1]])
=> [LexAddr a 0 0]
```

אם המשתנה אינו מופיע במאגר הכתובות, יחזור FreeVar.

הפונקציה הראשית להוספת הכתובות בביטוי נתון:

```
export const addLexicalAddresses = (exp: CExpLA): Result<CExpLA> => {
```

```
  const visitProc = (proc: ProcExpLA, addresses: LexicalAddress[]): Result<ProcExpLA> => {
    let newAddresses = crossContour(proc.params, addresses);
    return bind(mapResult(b => visit(b, newAddresses), proc.body),
              (bs: CExpLA[]) => makeOk(makeProcExpLA(proc.params, bs)));
  };
  const visit = (exp: CExpLA, addresses: LexicalAddress[]): Result<CExpLA> =>
    isBoolExp(exp) ? makeOk(exp) :
    isNumExp(exp) ? makeOk(exp) :
    isStrExp(exp) ? makeOk(exp) :
    isVarRef(exp) ? makeOk(getLexicalAddress(exp, addresses)) :
    isFreeVar(exp) ? makeFailure(`unexpected LA ${exp}`) :
    isLexicalAddress(exp) ? makeFailure(`unexpected LA ${exp}`) :
    isLitExp(exp) ? makeOk(exp) :
    isIfExpLA(exp) ?
   safe3((test: CExpLA, then: CExpLA, alt: CExpLA) => makeOk(makeIfExpLA(test, then, alt)))
                (visit(exp.test, addresses), visit(exp.then, addresses), visit(exp.alt, addresses)) :
    isAppExpLA(exp) ?
     safe2((rator: CExpLA, rands: CExpLA[]) => makeOk(makeAppExpLA(rator, rands)))
              (visit(exp.rator, addresses), mapResult(rand => visit(rand, addresses), exp.rands)) :
    isProcExpLA(exp) ? visitProc(exp, addresses) :
    exp;
  return visit(exp, []);
};
```

לחמש הפעולות שהדגמנו יש תבנית משותפת:

- פעולות על עץ תחבירי מופשט, בהתאם לבעיה הספציפית:
  - מקרה קצה
    - חישוב גובה: כל קודקוד מגדיל את הגובה באחד
    - האם משתנה חופשי בביטוי: VarRef - כן
    - מציאת התייחסות למשתנים: VarRef - הרשימה [v]
    - המרת let: החלפת ה LetExp ב AppExp
    - הוספת כתובת לקסיקאלית:
      - VarDecl (lambda, define, let) – הרחבת מאגר ההגדרות
      - VarRef – החלפת ה VarRef ב LexicalAddrees/FreeVar בהתאם למאגר
  - קריאה רקורסיבית לתתי הביטויים
    - מיזוג תוצאות
      - חישוב גובה: max, +
      - האם משתנה חופשי בביטוי: or
      - מציאת התייחסות למשתנים: union
      - המרת let: בניית קודקוד תואם חדש עם תתי הביטויים המומרים

- הוספת כתובת לקסיקאלית: בניית קודקוד תואם חדש עם תתי הביטויים החדשים הכוללים כתובות

2.4 הגדרה פורמלית של סמנטיקה ומימושה באינטרפרטר

עד כה, הגדרנו באופן פורמאלי את התחביר עבור השפות L1-L3, ומימשנו אותו ע"י כתיבת Parser. כעת, נגדיר את הסמנטיקה של שפות אלו – כלומר, כיצד המבנים/ביטויים השונים בשפה הופכים לערכים, ונממש סמנטיקה זו כאינטרפרטר:

- נגדיר את סוגי/טיפוסי הערכים עבור הביטויים השונים בשפה.
- נגדיר חוקי חישוב פורמאליים, המתארים מהו הערך עבור כל סוג של ביטוי.
- מימוש את חוקי החישוב באינטרפרטר

2.4.1 הגדרת ערכים וחוקי חישוב לשפות L1-L3

1. הערכים בשפות L1-L3

יש להגדיר לכל אחד מסוגי הביטויים בשפות את הערך שאליו הוא יחושב.

1L

ביטויים אטומיים: מספרים, בולאניים, אופרטורים פרימיטיביים, משתנים.
ביטויים מורכבים: define, הפעלה של אופרטור.

🡸 נגדיר את הערכים הבאים עבורם:

Value = Number | Boolean | PrimOp [*] | undefined[**]

נקודות לדיון:

* מה הערך של ביטוי אופרטור פרימיטיבי

לדוגמא: מה הערך של - ,+?

גישה א: ערכו של אופרטור פרימיטיבי זהה לערכה של פרוצדורת משתמש

- כזכור (כפי שנראה להלן ב 2L), הערך של הגדרת פרוצדורה (ProcExp) הוא Closure.
- באותו אופן, האינטרפרטר יגדיר מראש כל אופרטור פרימיטיבי כ Closure ויוסיף אותו לסביבה, ויתן לו שם (כמו הגדרת פרוצדורת משתמש ע"י define)
- בגישה זו, האופרטור הפרימיטיבי ייוצג בתחביר כ VarRef, כמו שמות של פרוצדורות המוגדרות על ידי כותבי הקוד.
- ביצוע/חישוב של האופרטור, יתחיל בגישה לסביבה עם שמו כדי לקבל את ה Closure שלו.

גישה זו תמומש בתרגול הבא.

<u>גישה ב</u>: האופרטור הפרימיטיבי מטופל באופן מיוחד באינטרפרטר

- יש לו ייצוג מיוחד בתחביר (כ PrimOp בפארסר שלנו, לא כ'סתם' VarRef)
- הערך שלו הוא מיוחד, PrimOp
- כאשר נדרש להפעילו, יופעל קוד מיוחד באינטרפרטר עבור כל סוג של אופרטור פרימיטיבי.

בהרצאות נלך על הגישה השניה.

** מה הערך של define

Define הוא גוף חריג ב'נוף' של השפה הפונקציונאלית. אין לו למעשה ערך, אלא רק side-effect. מאחר שבשפה פונקציונאלית חייב להיות ערך לכל ביטוי, נגדיר לו ערך באופן מלאכותי.

הבחירה הטבעית: נגדיר טיפוס בשם Void

```
interface Void { tag: "void" }
makeVoid
isVoid
```

השימוש ב void גורר בעיות טכניות ב JS, לכן נבחר ב undefined של JS.

<u>2L</u>

נוספו שני סוגי ביטויים: IfExp, **ProcExp**

Value = Number | Boolean | PrimOp | undefined | **Closure**

<u>L3</u>

נוסף המבנה LetExp, וכן הגדרת **זוגות ורשימות, סמלים ומחרוזות**.
יש להגדיר ערך המייצג זוגות רשימות סמלים ומחרוזות.

באופן מעשי, ערך זה מייצג כל סוג של <u>היררכיית ערכים</u>.

ניתן לייצג כל היררכיית ערכים בעזרת הקונספט של S-Exp (בפארסר ה S-Exp ייצג את היררכיית הטוקנים בתוכנית, כאן הוא מייצג את היררכיית הערכים).

ה S-Exp הוא ה literal expression של השפות שלנו, מאפשר להגדיר כל קומבינציה של ערכים.

SExp = Number | Boolean | Symbol | String | Pair(SExp) | List(SExp)

Value = PrimOp | undefined | Closure | SExp

2. הגדרה פורמאלית של חוקי החישוב

כזכור, המשמעות/הסמנטיקה של השפה היא האופן שבה ביטויים בתוכנית הופכים לערכים ('סמנטיקה אופרציונאלית).
את התחביר הגדרנו באופן פורמלי ע"י ניסוח חוקים לקסיקאליים (שהגדירו את המילים בשפה) וחוקים תחביריים (שהגדירו את המבנים בשפה). באופן דומה, נגדיר את הסמנטיקה על ידי חוקי חישוב, המקובעים כיצד הופך כל אחד מהביטויים התחביריים האפשריים בשפה לערך:

Atomic Expressions

Numbers
eval(<num-exp> exp, env) ➔ exp.val
Example: NumExp(3) ➔ 3

Booleans
eval(<bool-exp> exp, env) ➔ exp.val
Example: BoolExp(#t) ➔ true

Primitive operators
eval(<prim-op> exp, env) ➔ exp
Example: PrimOp("+") ➔ PrimOp("+")

Var references
eval(<var-ref> exp, env) ➔ apply_env(env, exp.var)
Example: VarRef("x") Env({ x : 3, y : 4}) ➔ 3

Compound Expressions

Define expressions
eval(<def-exp exp>, env) ➔
    val = eval(exp.val,env)
    extend_env(env, exp.var, val)
    return undefined
Example: DefExp(x, AppExp(+ 5 6)), Env({y : 8}) ➔ undefined, Env({ y:8, x:11})

If expressions
eval(<if-exp exp>, env) ➔
    test = eval(exp.test, env)
    test ? eval(exp.then, env) : eval(exp.alt, env)
Example: IfExp(AppExp(> x 4), NumExp(5), NumExp(6)), Env({ y:8, x:11}) ➔ 5

Literal expression
eval(<lit-exp exp>, env) ➔ exp.val
Example: LitExp('(1 2 3)) ➔ Sexp(List(1,2,3)); LitExp('abc) ➔ Symbol(abc)

Procedures
eval(<proc-exp exp>, env) ➔ make_closure(exp.args, exp.body)
Example: ProcExp([x],[AppExp(* x x)] ➔ Closure([x],[AppExp(* x x)])

Application Expressions
eval(<app-exp exp>, env) ➔
  proc = eval(exp.rator,env)
   args = [eval(rand,env) for rand in exp.rands]
   is_primitive(proc) ? apply_primitive(proc,args) : apply_closure(proc,args,env)

הפרוצדורה apply_primitive באינטרפרטר מממשת, בשפה של האינטרפרטר, את הסמנטיקה של האופרטורים הפרימיטיביים השונים בשפה L.

Example: AppExp((IfExp((> 3 4) - +), [NumExp(3), NumExp(4)]) ➔ 7

הפרוצדורה apply_closure באינטרפרטר, מחשבת את הערך של הפעלת הפרצודרה הנתונה ב L עם הארגומנטים (המחושבים) הנתונים.
קיימים (לפחות) שני מודלים לחישוב זה:

- מודל ההצבה (substitution model)
- מודל הסביבות (environment model)

נתמקד השבוע במודל ההצבה.

<u>מודל ההצבה</u> - כדי לחשב פרוצדורה נתונה (Closure) על פי רשימת ארגומנטים:

- מציבים ב body של הפרוצדורה את הארגומנטים, כל אחד מהם יחליף את כל ה VarRef שלו ב body.
- מחשבים את ה body (כלומר, קוראים ל eval עבור כל הביטויים ב body, ומחזירים את הערך של הביטוי האחרון)

Example: AppExp(ProcExp([x], [AppExp(* x x)]), [3]) ➔ 9

<u>הערה</u>: בחוק החישוב שהצגנו עבור הפעלת פרוצדורה/אופרטור, חישבנו מראש את כל הארגומנטים. גישה זו מכונה applicative order. בהמשך נדון בגישה אחרת, המשהה חישוב זה של הארגומנטים עד לרגע שבו זה נדרש, גישה המכונה normal order

2.4.2 מימוש האינטרפרטר

1. תשתית

הגדרת הערכים השונים: L3-value.ts

מימוש הסביבה: L3-env.ts

2. קוד האינטרפרטר

המסגרת הראשית:

L3-eval.ts

evalL3program
חישוב הערך של התוכנית, הוא חישוב הערך של כל הביטויים בגוף התוכנית, והחזרת הערך של הביטוי האחרון.

evalSequence
חישוב סדרת ביטויים, העשויה להתחיל בסדרת ביטויי define, מחשבת אותם בזה אחר זה, ומחזירה את הערך של הביטוי האחרון.

evalDefineExps
חישוב סדרת ביטויים המתחילה בביטוי מסוג define.
הרחבת הסביבה ע"פ ביטוי define הראשון, וחישוב שאר הביטויים ברשימה ע"פ הסביבה המורחבת.

L3applicativeEval
חישוב ביטוי בעל ערך (Cexp) ע"פ הסוג שלו, בהתאם לחוקי החישוב.

חישוב הפעלת אופרטור פרימיטיבי:

evalPrimitives.ts
applyPrimitive
מימוש ב TS של כל אחד מהאופרטורים הפרימיטיביים ב 3L

חישוב הפעלת קלוז'ר:

פונקציה ראשית

L3-eval.ts
applyClosure

- המרת רשימת הערכים להצבה לרשימת ביטויים מקבילה (value2LitExp). נדרש משיקולי תאימות טיפוסים: גוף הפרוצדורה להצבה מוגדר במושגים של CExp של הפארסר, בעוד שהערכים להצבה הם כבר במושגי ה Valueשל האינטרפרטר.

מימוש ההצבה

substitute.ts
substitute

הצבת רשימת ביטויי ערכים ברשימת משתנים, עבור רשימת ביטויים.

sub

הצבת רשימת ביטויי ערכים ברשימת משתנים, עבור ביטוי נתון:

- מקרי קצה
  - VarRef: החלפתו בביטוי המתאים (אם נדרש)
  - ProcExp: סינון רשימת המשתנים להצבה, כך שלא תכלול את המשתנים של הפרוצדורה המקוננת.

```
(L3
 (define a 3)
 (define b 4)
 (
  (lambda (x)
   (if x
      ( (lambda (x) (* a x)) 3)
      ( (lambda (x) (+ b x)) 4)
   )
  )
  #t
 )
)

<program
 [
   <def-exp <var-decl a>  <num-exp 3>>
  <def-exp <var-decl b>  <num-exp 4>>
  <app-exp
    <proc-exp
     [<var-decl x>]
     [<if-exp <var-ref x>
            <app-exp
              <proc-exp <var-decl x> [app-exp <prim-op *> [<var-ref a> <var-ref x>]]>
              <num-exp 3>
            >
            <app-exp
              <proc-exp <var-decl x> [app-exp <prim-op +> [<var-ref b> <var-ref x>]]>
              <num-exp 4>
            >
    >]
   >
   <bool-exp #t>
  >
 ]
>
```

```
ENV: [ a:3, b:4]

args = [true]
vars = [x]
litArgs = [<bool-exp #t>]

argNames = [x]
subst = [<x <bool-exp #t>>]
freeSubst = []
```

דוגמא נוספת:

```
(define z 3)
(
  (lambda (f)                    ;; rator
    (lambda (z)
      (f z)
    )
  )
  (lambda (w) (+ w z))    ;; rand
)

-->
(define z 3)
(lambda (z)
      ((lambda (w) (+ w z)) z)
)
```

בעיה: ההצבה מטשטשת את ההבדל בין z בפרוצדורה עם הפרמטר w, הקשור ל z ב define, ובין ה zבפרוצדורה הפנימית שמקבלת z.

פתרון: נדאג לכך שכל שמות המשתנים בפרוצדורות שונות (הקריאה לפרוצדורה renameExps במסגרת ה applyClosure)

```
-->
(define z 3)
(lambda (z1)
      ((lambda (w2) (+ w2 z)) z1)
)
```

2.4.3 Normal Order

ראינו כי בגישת Applicative Order מחשבים את הערך של כל האופרנדים לפני הפעלת הפרוצדורה. ניתן לחשוב על גישה אחרת שבה לא במחשבים בשלב זה את הארגומנטים, אלא מציבים אותם כפי שהם כרגע. החישוב של כל ארגומנט ייעשה בהמשך רק כאשר נזדקק לו. גישה זו מכונה Normal Order.

דוגמאות:

א.

```
(
 (lambda (x) (+ x x))
 (* 2 3)
)
```

אפליקטיב: נחשב תחילה את (3 2 *) ונציב

```
(+ 6 6)
```

נורמל: נציב (3 2 *) מבלי לחשבם בשלב זה. הם יחושבו מאוחר יותר כאשר זה יידרש עבור הפעלת האופרטור הפרימיטיבי +.

```
(+ (* 2 3) (* 2 3))
```

בדוגמא זו, אפליקטיב יעיל יותר, בנורמל החישוב של (3 2 *) יבוצע פעמיים.

ב.

```
(
  (lambda (x y z) (if x y z))
  #t 3 (\ 30 3 2)
)
```

אפליקטיב: כל שלושת האופרנדים מחושבים מראש

```
(if true 3 5)
```

נורמל: האופרנדים מוצבים כפי שהם, החישוב נדחה לזמן בו האופרנד נדרש

```
(if #t 3 (\ 30 3 2))
```

במקרה זה נורמל יעיל יותר, אין צורך מעשי לחשב את הביטוי של ה alt (2 3 30 \)

ג.

```
(
  (lambda (x y z) (if x y z))
  #t 3 (\ 30 0)
```

)

אפליקטיב: כל שלושת האופרנדים מחושבים מראש, כך שהתוכנית תיפול על חלוקה באפס בחישוב האופרנד השלישי.

נורמל: האופרנדים מוצבים כפי שהם, החישוב נדחה לזמן בו האופרנד נדרש. מאחר שבתרחיש זה אין צורך לחשב אתת ה alt, לא נגיע לחלוקה באפס.

```
(if #t 3 (\ 30 0))
```

ד.

```
(define loop (lambda (x) (loop x)))
(define f (lambda (x) 5))
(f (loop 0))
```

אפליקטיב: יחושב האופרנד (loop 0) בהפעלה של f, וניכנס ללואה אינסופית.

נורמל: האופרנד (loop 0) יוצב ב f מבלי לחשבו. מאחר ואין התייחסות אליו ב f, הוא לא יחושב כלל והתוכנית תסתיים כסדרה.

<u>שקילות</u>:

האם חישוב ביטוי ב applicative order וב normal order שקול?
ראינו שלא – דוגמאות ג,ד

משפט צ'רץ'-רוזר: אם חישוב ביטוי ב applicative order מסתיים וללא שגיאת זמן ריצה, אז חישובו ב normal order שקול. כלומר, סדר החישוב לא משנה במקרה זה (כאשר אין כמובן side-effects)

<u>מימוש</u>: L3-normal.ts

שינויים:

- במקרה של IsApp בפרוצדורה L3normalEval, לא מחשבים את האופרנדים
- מחשבים את האופרנדים לפני הקריאה ל applyPrimitive בפרוצדורה L3normalApplyProc.
- אין צורך בהמרת האופרנדים בחזרה מערכים לביטויים במקרהשל isClosure בפרוצדורה L3normalApplyProc (value2LitExp) כי הם לא מחושבים (כלומר הטיפוס שלהם הוא CExp ולא Value עדיין).

2.5 מודל הסביבות

2.5.1 מוטיבציה

במודל ההצבה, כל הפעלה של פרוצדורה (גם אם היא הופעלה כבר בעבר), כרוכה בשכתוב גוף הקוד שלה:

- חישוב ערכי הארגומנטים (באפליקטיב, או אח"כ בנורמל)
- שינוי שמות כל הפרמטרים של כל הפרוצדורות המוגדרות ב body
- (באפליקטיב) החזרת ערכי הארגומנטים למבנה של ביטויים.
- הצבת הארגומנטים בכל VarRef מתאים.

← כבד!

מה קורה בשפות כמו Java, C++?

- נפתח Activation Frame, בו מוגדרים הערכים של כל הפרמטרים של הפרוצדורה, מבלי לשנות הקוד.
- כל פעם שיש התייחסות למשתנה, ניגשים למקום ב AF שבו הוא מוגדר (כלומר הוא נקרא מהזיכרון)
- קיימת היררכיה של פריימים, כלומר ניתן לגשת רק ל AF האחרון, כך שאין בלבול בין שמות זהים של משתנים שונים.
- בתום הפרוצדורה ה AF נסגר ועוברים ל AF הקודם.

← ניישם קונספט דומה אך שונה עבור האינטרפרטר של שפות ה-L: מודל הסביבות.

2.5.2 הגדרת מודל הסביבות

1. מבנה נתונים

עד כה: השתמשנו בסביבה הגלובלית, בו אוכסנו כל ה bindings שהוגדרו ע"י פעולות ה define (במימוש הפונקציונאלי, הסביבה היתה למעשה רשימה מקושרת של 'סביבות'/'פריימים', כאשר כל סביבה היא מעין פריים עם בינדינג אחד)

- נרחיב קונספט זה לכדי היררכיה של פריימים (שהשורש שלה הוא הסביבה הגלובלית).
- בכל הפעלה של פרוצדורה, במקום לשכתב אותה כמו במודל ההצבה, נגדיר פריים חדש, שבו יוגדרו ה bindings של הפרמטרים של הפרוצדורה עם הערכים שנשלחו עבורם. פריים זה יתווסף להיררכיה, כלומר יקושר לאחד הפריימים הקיימים.
- כאשר יש VarRef, האינטרפרטר יחפש את ערכו בהיררכיית פריימים, מהפריים האחרון ולמעלה.

כלומר:

- השימוש במבנה הנתונים של הסביבה משחרר אותנו מהצורך בהצבה של הארגומטים בתוך גוף הפרוצדורות.
- המבנה ההיררכי של הפריימים משחרר אותנו מהצורך בשינוי שמות המשתנים לייחודיים. המופע הראשון של הגדרת המשתנה בהיררכיית הפריימים הוא הרלבנטי לקטע הקוד הנתון.

הערת מינוח: נשתמש במונח 'פריים' עבור מבנה הנתונים הכולל bindings לשם חישוב פרוצדורה, ובמונח 'סביבה' עבור רצף כלשהו של frames.

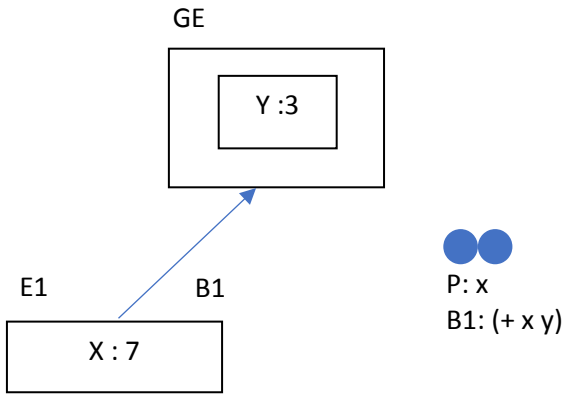
נעדכן את חוק החישוב של הפעלת הפרוצדורה, כך שיתאים למודל הסביבות:

```
Procedure/Operator application  ;; (+ 3 4)  ;;  ((lambda (x) (* x x)) 3)
eval(<app-exp exp>, env) ➔
   proc = eval(exp.rator,env)
   args = [ eval(r,env) for r in exp.rands ]
   is_primitive(proc) ? apply_primitive(proc,args) :
                              eval_sequence(proc.body,
                                   ext_env(env,make_frame(proc.params, args)))
```

דוגמאות:

א.

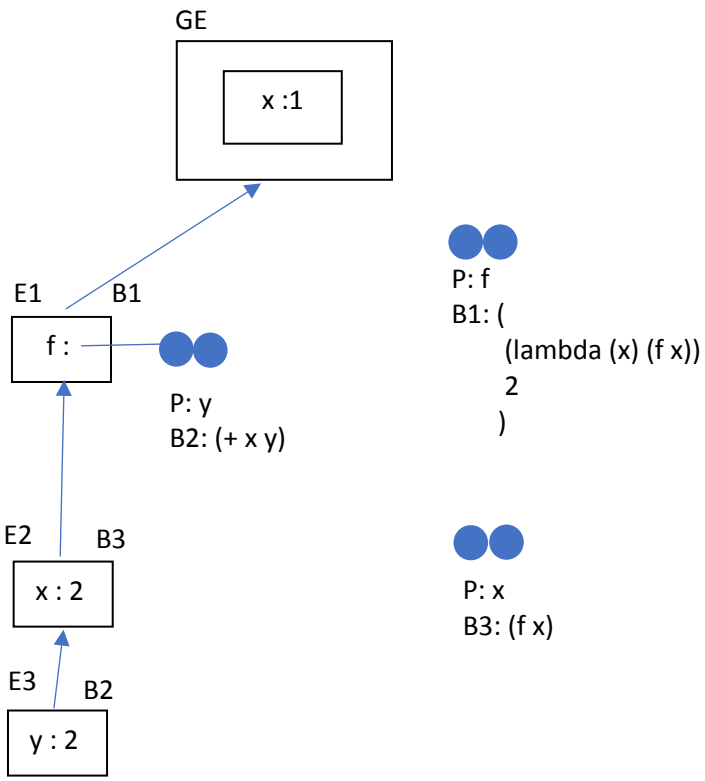
```
(define y 3)
(
 (lambda (x) (+ x y))
 (+ 5 2)
)
```



ב.

```
(define x 1)
(
  (lambda (f)
   (
    (lambda (x) (f x))
```

```
    2
    )
   )
  (lambda (y) (+ x y))
)
```

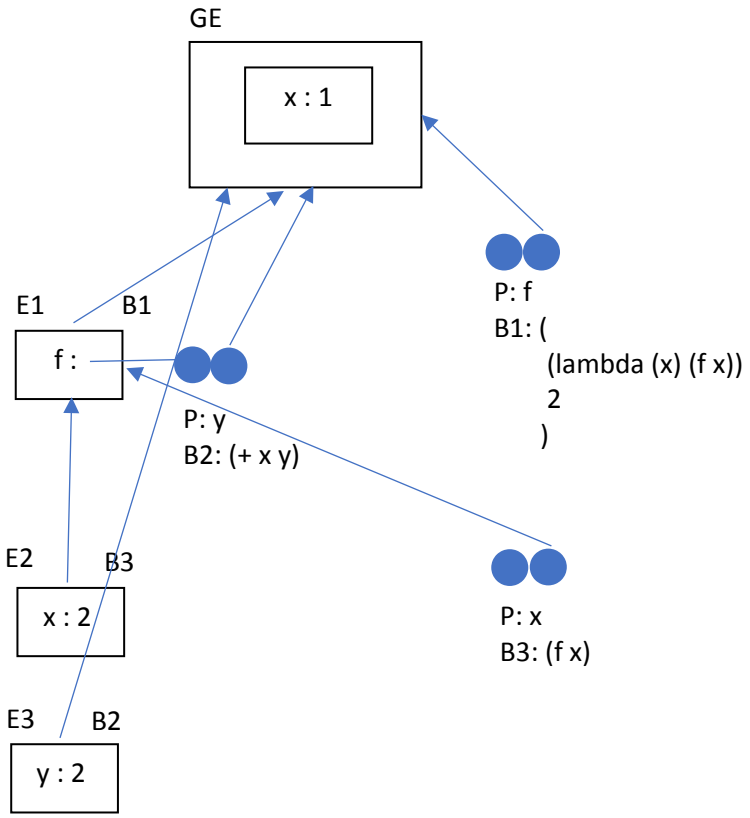


Value: 4 (instead of 3)

האופן שבו בנינו את היררכיית הפריימים – חיבור הפריים החדש של הפעלת פרוצדורה לסביבה הנוכחית של <u>זמן ההפעלה</u>, כלומר לפריים האחרון – גרמה לכך, שהפרמטר x של הפרוצדורה (lambda (y) (+ x y)) התפרש <u>כערך של x בזמן הפעלת הפרוצדורה ולא בזמן שהפרוצדורה נוצרה</u>.
(במודל ההצבה, עניין זה בדיוק גרם לנו לשינוי שמות המשתנים בפרוצדורות כך שיהיו ייחודיים)

← נרחיב את מבנה ה Clusore כך שמלבד רשימת הפרמרטרים וה body, נזכור גם את 'העולם' שבו נוצרה הפרוצדורה, כלומר מה היתה הסביבה כאשר היא נוצרה.
כאשר פרוצדורה תופעל, הפריים החדש עם הצבות הפרמטרים <u>יחובר לסביבה שהיתה בזמן שהיא נוצרה</u>.

```
(define x 1)
(
  (lambda (f) ( (lambda (x) (f x)) 2 )))
  (lambda (y) (+ x y))
)
```



Value: 3
(correct)

<u>עדכון חוקי</u>

<u>החישוב</u>:

Procedures ;; (lambda (x) (* x x))
eval(<proc-exp exp>, env) ➔ make_closure(exp.args, exp.body**, env**)

Procedure/Operator application ;; (+ 3 4) ;; ((lambda (x) (* x x)) 3)
eval(<app-exp exp>, env) ➔
  args = [eval(r,env) for r in exp.rands]
  proc = eval(exp.rator,env)
  is_primitive(proc) ? apply_primitive(proc,args) :
                eval_sequence(**proc.body**,

ext_env(**proc.env**,make_frame(proc.params, args)))

'גישה' זו, של הפעלת פרוצדורה לאור הסביבה שהיתה בזמן הגדרתה ולא הסביבה הקיימת בזמן הפעלתה, מכונה lexical scoping.
[במשך כמה שנים בשנות השבעים לא עמדו על נקודה זו – האינטרפרטר של Lisp עבד עם dynamic scoping, כלומר חיבר את הפריים של הפעלת הפרוצדורה לסביבה בזמן הפעלתה]

כזכור, אין צורך לטפל באינטרפרטר בצורת ה let, כי היא סה"כ קיצור תחבירי להפעלה של פרוצדורה.
בכל זאת, כדי לחדד את עניין הסביבות (בדומה להפעלת פרוצדורה, גם ב let נדרש לפתוח frame חדש עבור חישוב ה body עם המשתנים הלוקאליים), נגדיר עבורה חוק חישוב ונממש אותו באינטרפרטר:

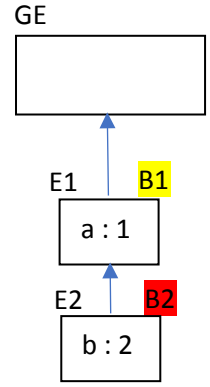
```
Let expression  ;; (let ((a 3) (b 4)) (+ a b))
eval(<let-exp exp>, env) ➔
   vars = vars in exp.bindings
   vals = vals in exp.bindings
   cvals = [ eval(val,env) for val in vals]
   eval_sequence(exp.body, ext_env(env, make_frame(vars,cvals))
```

[במודל ההצבה השורה האחרונה בחוק היתה:
[eval_sequence(**substitute(rename(**exp.body**))**, **env**)

שימו לב שאת הפריים החדש עם ערכי המשתנים הלוקאליים מחברים לסביבה הנוכחית – סביבת ההגדרה של גוף ה let הוא הסביבה הנוכחית.

דוגמא:

```
(let ( (a 1) )
  (let ( (b (+ a a )) )
     (+ a b)))
```



<u>הערה</u>: השוואה למודל ה Activation Frame ב C/Java

- ההיררכיה ב C/Java היא שרשור של פריימים, כאן מדובר בעץ (ראינו כי הפריים של הפעלת פרוצדורה לא מתחבר לפריים האחרון אלא לזה שהיה כאשר הפרוצדורה הוגדרה).

- במודל הסביבות ניתן להתייחס להגדרות של משתנה בפריימים אחרים לאורך ההיררכיה. ב Java/C ניתן לגשת רק לפריים הנוכחי.
- עם תום הפעלת הפרוצדורה ה AF נמחק ב Java/C. במודל הסביבות הפריים נשאר כל עוד יש אליו התייחסות (בפרט, כל עוד קיימת פרוצדורה שנוצרה תחתיו)

2.5.3 עדכון קוד האינטרפרטר עבור מודל הסביבות

תשתית:
מימוש מבנה הנתונים של הסביבות: L4-env.ts
עדכון הערכים (כעת מבנה ה Closure כולל שדה נוסף המציין את הסביבה שבה הוא הוגדר): L4-value.ts

האינטרפרטר: L4-eval.ts

evalProc – הוספת הסביבה הנוכחית לקלוז'ר המוחזר
applyClosure4 – מימוש חוק החישוב עבור הפעלת פרוצדורה (בניית פריים וחיבורו לסביבת הגדרתה)
evalLet4 – מימוש חוק החישוב עבור let (הגדרת פריים עם המשתנים הלוקאליים וחיבורו לסביבה הנוכחית)

2.5.4 Object Oriented Programing in L2

אמרנו כבר כי ניתן לממש OOP החל מ-2L. נדגים זאת כעת (כאשר ניתן להמחיש זאת ויזואלית עם דיאגרמת הסביבות).

דוגמא: מחלקה הכוללת ערך מספרי, ומתודה של חיבורו למספר נתון.

```
class Adder {
  int a;
  Adder(int a) { this.a = a; }
  public int add(int x) { return a + x; }
}

Adder a3 = new Adder(3);
Adder a5 = new Adder(5);
a3.add(2);
→ 5
a5.add(2);
→ 7
```

מימוש ב-2L:

הגדרת בנאי, המחזיר מופע של מחלקה זו:

```
(define make-adder
```

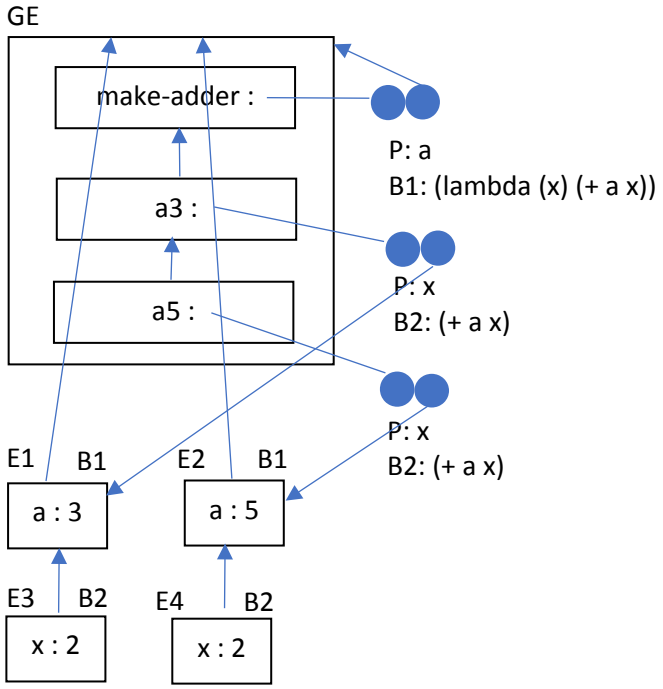
```
(lambda (a)
  (lambda (x)
    (+ a x))))
```

הגדרת שני מופעים של המחלקה (אחד עם שדה בעל ערך 3, ואחד עם שדה בעל ערך 5):

```
(define a3 (make-adder 3))
(define a5 (make-adder 5))
```

ביצוע המתודה עבור כל אחד מהאובייקטים:

```
(a3 2)
→ 5
(a5 2)
→
```




```
(define make-adder
  (lambda (a)
    (lambda (x)
      (+ a x))))

(define a3 (make-adder 3))

(define a5 (make-adder 5))

(a3 2)

(a5 2)
```

דוגמא נוספת: המחלקה Pair

```
class Pair {
  int a,b;

  Pair(int a, int b) { this.a = a; this.b = b; }

  int getFirst() { return a; }
  int getSecond() { return b; }
  int add() { return a + b;}
  Pair scale (int k) { return new Pair(a*k,b*k) }
}
```

מימוש ב 2L:

```
(define make-pair
  (lambda (a b)
    (lambda (method)
      (if (eq? method 'first) a
         (if (eq? method 'second) b
            (if (eq? method 'add) (+ a b)
               (if (eq? method 'scale)
                  (lambda (k)
                    (make-pair (* a k) (* b k)))
                  #f)))))))

(define p (make-pair 1 2))
(p 'first)  → 1
(p 'second) → 2
(p 'add) → 3
(((p 'scale) 2) 'add) → 6
```

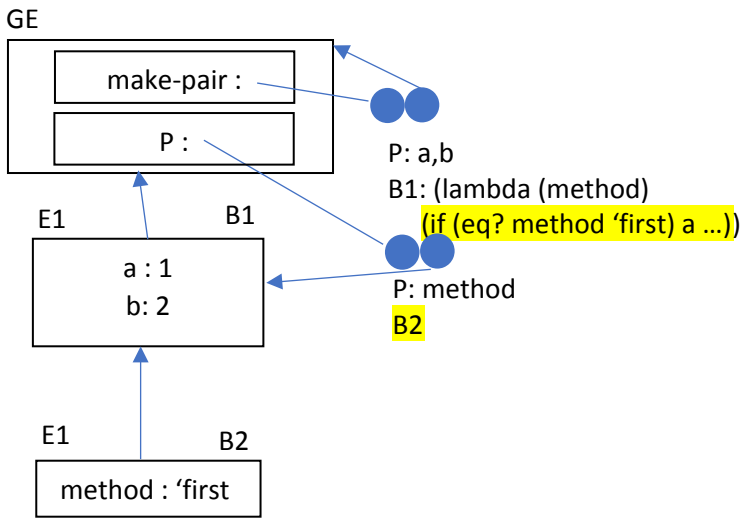
ניתן להציע גם מימוש כללי יותר, בו 'המשתמש' שולח את המתודה/הפעולה שהוא רוצה לבצע על האובייקט:

```
(define make-pair
  (lambda (a b)
    (lambda (f)
      (f a b))))

(define p (make-pair 1 2))
```

```
(p (lambda (a b) a))
→ 1
(p (lambda (a b) b))
→ 2
(p (lambda (a b) (+ a b)))
→ 3
(p (lambda (a b) (* a b)))
→ 2
(p (lambda (a b) (make-pair (* 2 a) (* 2 b))))
```

```
(define make-pair
 (lambda (a b)
  (lambda (method)
   (if (eq? method 'first) a
       (if (eq? method 'second) b
       (if (eq? method 'add) (+ a b)
           (if (eq? method 'scale)
```



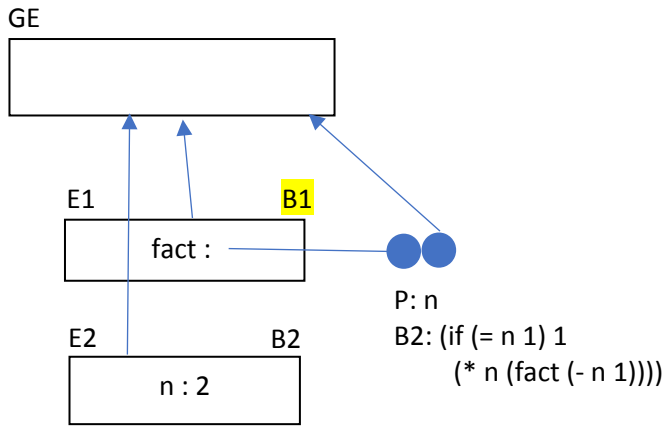
<u>הערה</u>: החל מהשפה 3L ניתן לממש OOP גם בעזרת רשימות, כאשר האיבר הראשון ברשימה הוא סימבול המציין את שם המחלקה (כמו ה tag בממשקים שלנו באינטרפרטר, ‘pair’), ושאר אברי הרשימה הם ערכי השדות. נשתמש בגישה זו כאשר נכתוב אינטרפרטר לפרולוג בשפה 5L.

2.5.5 רקורסיות

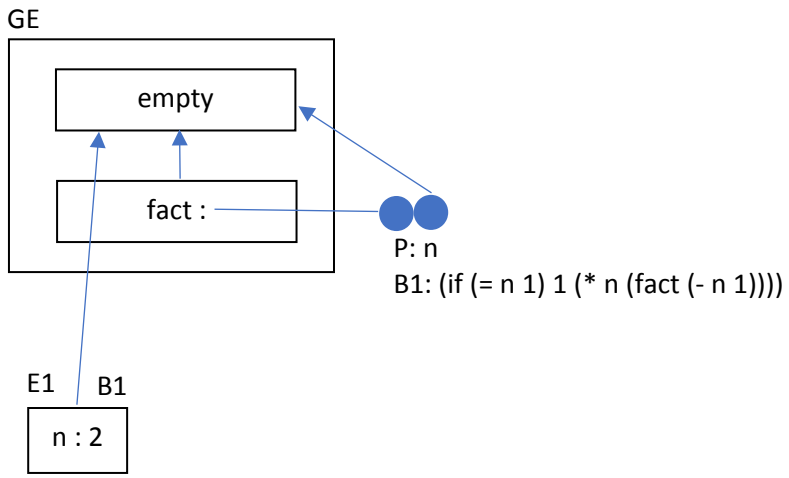
מודל הסביבות, כפי שהוצג עד כה, אינו תומך ברקורסיות (מודל ההצבה תומך ברקורסיות החל מ 2L).

דוגמא: הפרוצדורה fact בשני אופנים של הגדרה: let, define

```
(let ((fact (lambda (n)
              (if (=n 1)
                  1
                  (* n (fact (- n 1))))))
(fact 2))
```




```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
(fact 2)
```

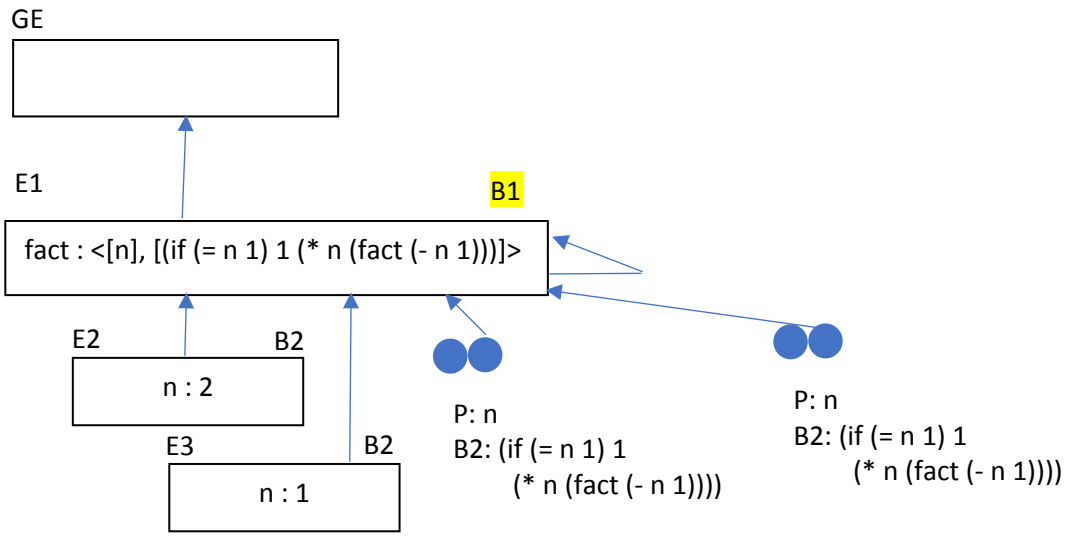


פיתרון א: נשהה את יצירת הקלוז'ר במקרים כאלה עד לאחר יצירת הפריים שבו המשתנה שלו מוגדר.

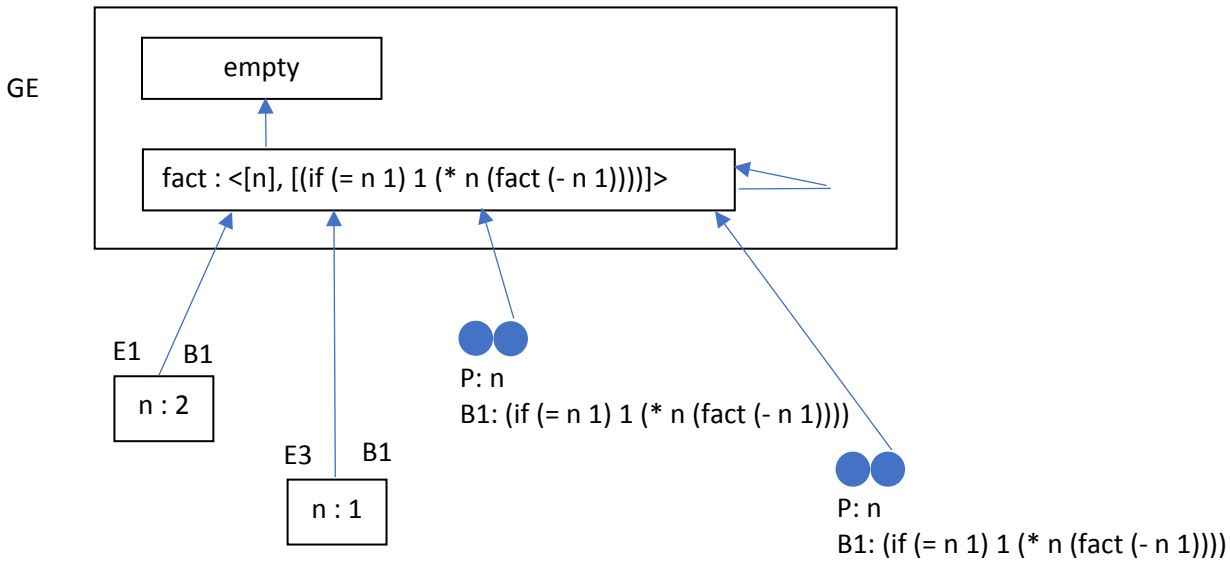
נמקד תרחיש זה על ידי הגדרת צורה חדשה בשפה (4L) **letrec** - וראיציה של let עבור משתנים לוקאליים מסוג פרוצדורות רקורסיביות, בה נשמר בפריים מבנה נתונים עם נתוני הפרוצדורה (מערך הפרמטרים ומערך הביטויים ב body) אך לא ה closure עצמו.
לשם כך נגדיר פריים מסוג חדש 'rec env', בו טיפוס הערכים ב bindings אינו Value אלא מערך VarDecl ומערך CExp (עבור הפרמטרים וביטויי גוף הפרוצדורה). בפריים זה הקלוז'ר של הפרוצדורה נבנה בכל פעם שמבוצע applyEnv, כלומר בכל פעם שנדרש להשתמש בקלוז'ר, כאשר הסביבה של הקלוז'ר היא הפריים שבה מאוחסנים נתוניה.

לדוגמא:

```
(letrec ((fact (lambda (n)
                 (if (=n 1)
                     1
                     (* n (fact (- n 1))))))
(fact 2))
```




```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
  (fact 2)
```



מימוש:

L4-env.ts - הגדרת RecEnv
L4-eval.ts - שכתוב evalLet כ evalLetRec ושכתוב evalDefineExps (עבור הגדרת משתנה המייצג פרוצדורה) כך שיגדירו פריים מסוג RecEnv במקום ExtEnv.

בנוסף, שכתוב evalDefineExps עבור מקרה שבו יש הגדרה של פרוצדורה:

```
const evalDefineExps = (exps: Exp[], env): Result<Value> => {
  let def = first(exps);
  if (isDefineExp(def)) {
    // Check if rhs is ProcExp - use a recEnv - else an extEnv
    if (isProcExp(def.val)) {
      const newEnv = makeRecEnv([def.var.var], [def.val.args], [def.val.body], env);
      return evalExps(rest(exps), newEnv);
    } else {
      const rhs = applicativeEval(def.val, env);
      if (isFailure(rhs)) {
        return rhs;
      } else {
        const newEnv = makeExtEnv([def.var.var], [rhs], env);
        return evalSequence(rest(exps), newEnv);
      }
    }
  } else {
    MakeFailure("never");
  }
}
```

חיסרון: בכל קריאה לפרוצדורה שהמשתנה המייצג אותה נמצא בפריים רקורסיבי, יש ייצור מחדש של אותו קלוז'ר שלה.

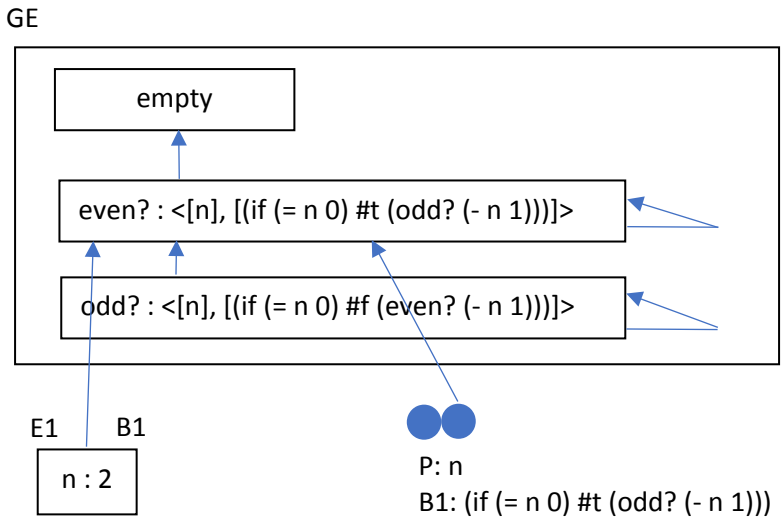
בעיה: קריאות הדדיות בין פרוצדרות לא נתמכות על ידי define

```
(define even?
  (lambda (n)
    (if (= n 0)
        #t
        (odd? (- n 1)))))

(define odd?
```

```
(lambda (n)
  (if (= n 0)
      #f
      (even? (- n 1)))))

(even? 2)
```



פתרון ב: נשתמש בפעולות השמה כדי לתמוך בסביבה גלובלית עם פריים אחד בלבד, וכן בפריים רקורסיבי הכולל ערכי קלוז'ר המחושבים פעם אחת.

הבעיה של הקריאה ההדדית נבעה מכך שהסביבה הגלובלית הוגדרה כרשימה מקושרת של פריימים, כדי להימנע מפעולות השמה בקוד האינטרפרטר.
נשנה את הגישה ונשתמש בפעולות השמה (גם עבור letrec), אך נעשה זאת באופן ממוקד ביותר:

- לפריים היחיד של הסביבה הגלובלית ניתן להוסיף binding חדש (נדרש כאשר מחשבים ביטוי define), אך לא ניתן לשנות binding קיים (לא נדרש).

- בפריים הרקורסיבי (הפריים של letrec) ניתן לשנות את הערך של binding קיים (נדרש לשם עדכון הקלוז'ר של הפרוצדורות הרקורסיביות לאחר שהפריים נוצר, כך שיצביעו לפריים זה), אך לא להוסיף binding חדשים (לא נדרש).

- נמקד תחבירית את פעולות ההשמה בממשק Box. רק דרך ממשק זה ניתן לבצע פעולות השמה.

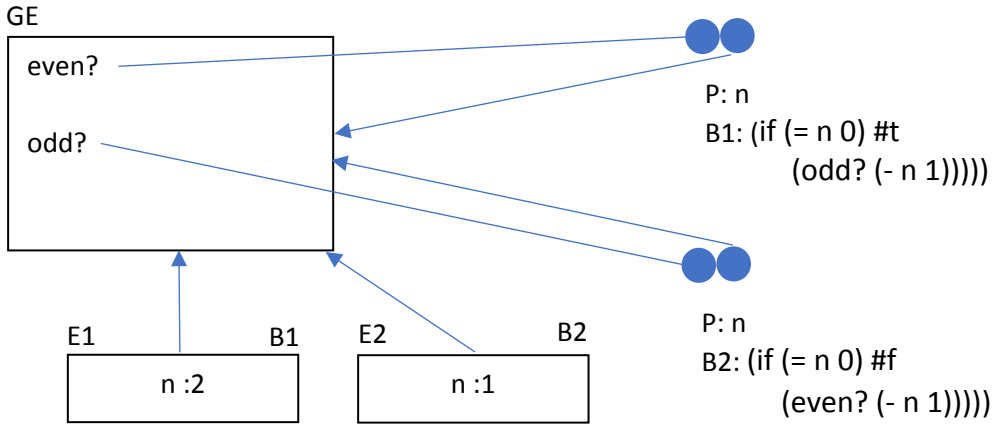
חוק החישוב של define:

- מחשבים את הערך של המשתנה החדש (כמו פעם)
- **מוסיפים את ה binding החדש** (המשתנה וערכו המחושב) **לפריים היחיד בסביבה הגלובלית**

```
(define even?
  (lambda (n)
    (if (= n 0)
        #t
        (odd? (- n 1)))))

(define odd?
  (lambda (n)
    (if (= n 0)
        #f
        (even? (- n 1)))))

(even? 2)
```

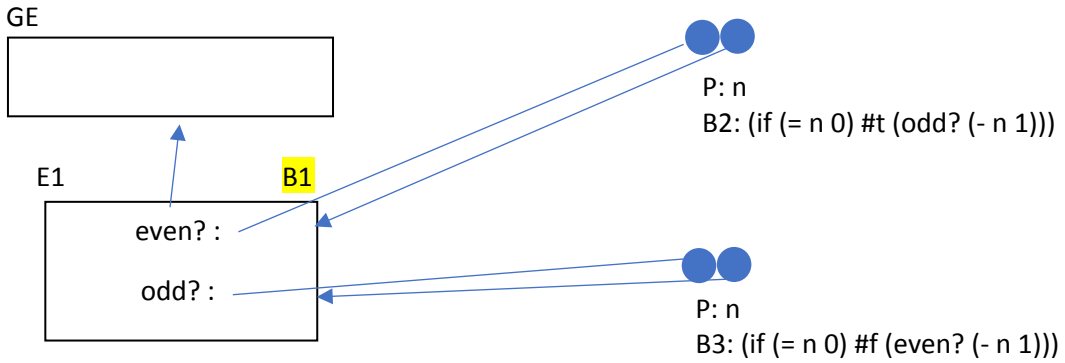


חוק החישוב עבור letrec:

- בניית פריים חדש, כאשר שמות המשתנים הלוקאליים מקושרים לערך undefined, וחיבורו לסביבה הקיימת.
- חישוב ערכי המשתנים כקלוז'רים, כאשר הסביבה הנוכחית כוללת כבר את הפריים החדש.
- עדכון ערכי המשתנים ב bindings של הפריים החדש עם החישוב של הערכים שלהם (קלוז'רים החיים בסביבה הכוללת את הפריים החדש).

```
(letrec ((even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))
         (odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))))

  (even? 2))
```



<u>מימוש</u>:

סביבה: L4-env-box.ts - הגדרת Box, הפעולות על הסביבה הגלובלית ועל הסביבה הרקורסיבית.

חישוב: L4-eval-box.ts

evalDefineExps –הוספת binding חדש לפריים היחיד בסביבה הגלובלית.

evalLetRec – הגדרת פריים רקורסיבי עם שמות המשתנים וערכי undefined, לאחר שהפריים נוצר יצירת קלוז'ר עם פריים זה לכל פרוצדורה ועדכון הערכים ב bindings של הפריים בהתאם (מבמקום ה undefined)

<u>הערה</u>: ניתן לממש רקורסיות במבנה let, define הרגילים (במודל הסביבות ובמודל ההצבה) בתבנית העיצוב הבאה:

```
(let ((fact (lambda (n fact)
              (if (= n 0) 1
                          (* n (fact (- n 1) fact))))))
   (fact 2 fact))

(define fact (lambda (n fact)
                (if (= n 0) 1
                            (* n (fact (- n 1) fact)))))
(fact 2 fact)
```

במודל ההצבה, ה define הרגיל תומך ברקורסיה כי בזמן ההפעלה הסביבה הגלובלית כוללת כבר את שם הפרוצדורה:

```
(define fact (lambda (n)
          (if (= n 0) 1
                      (* n (fact (- n 1))))))
(fact 2)
```

אך ה-let הרגיל לא, כי ההצבה משאירה ב-body את הקריאה הרקורסיבית לפונקציה, ששמה אינו מוכר בסביבה:

```
(let ((fact (lambda (n)
              (if (= n 0) 1
```

```
                    (* n (fact (- n 1)))))))
  (fact 2))
→
(
  (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))
  2
)
```

כיצד letrec תומך בקריאה הרקורסיבית במודל ההצבה?

...

3. טיפוסים

בפרק זה נרחיב את השפה L4 לשפה L5 כך שתכלול <u>טיפוסים</u>.

- תחביר
  - כיצד מתארים טיפוסים (מערכת טיפוסים)
  - כיצד מוסיפים טיפוסים לביטויים בתוכנית
- סמנטיקה (מה המשמעות של טיפוסים בתוכנית)
  - בדיקת תאימות טיפוסים, type checker
- מערכת להסק טיפוסים

3.1 תחביר

תחביר הטיפוסים מגדיר:

- מערכת הטיפוסים של השפה
- תיוג הטיפוסים של הביטויים בתוכנית

1. מערכת טיפוסים

מערכת טיפוסים (Type System) של שפה:

- קובעת את <u>הטיפוסים הפשוטים</u> של הביטויים/הערכים.
  לדוגמא: הטיפוסים הפשוטים ב TS, Java, C++ (int/number, boolean/bool,String/string/char*,...)
- קובעת כיצד ניתן להגדיר <u>טיפוסים מורכבים</u> מטיפוסים פשוטים.
  לדוגמא: הגדרת מחלקה ב Java/C++, איחוד/חיתוך/מכפלה קרטזית (מילון) של טיפוסים ב TS, ...
- מאפשרת להגדיר <u>יחס בין טיפוסים</u>.
  לדוגמא: יחס ירושה ב Java/C++, יחס הספציפיות בין מילונים ב JS

מערכת הטיפוסים של 5L:

- טיפוסים פשוטים
  number, boolean, string, void
- טיפוסים מורכבים
  ניתן רק להגדיר טיפוס מורכב של חתימת פרוצדורה (אין טיפוס של זוג, רשימה,...)
  [number * boolean -> void]

[Empty -> string]

...

- קביעת יחס בין טיפוסים

כרגע, לא ניתן לקבוע יחס בין טיפוסים (כמו לדוגמא לומר שטיפוס אחד יותר ספציפי מטיפוס שני). כלומר, כרגע, נדרשת התאמה מלאה בין טיפוסים.

הגדרה פורמלית של מערכת הטיפוסים של L5:

```
<texp>        ::= <atomic-te> | <composite-te> | <tvar>
<atomic-te>   ::= <num-te> | <bool-te> | <str-te> | <void-te>
<num-te>      ::= number   // num-te()
<bool-te>     ::= boolean  // bool-te()
<str-te>      ::= string   // str-te()
<void-te>     ::= void     // void-te()
<composite-te> ::= <proc-te>
<non-tuple-te> ::= <atomic-te> | <proc-te> | <tvar>
<proc-te>     ::= [ <tuple-te> -> <non-tuple-te> ] // proc-te(param-tes: list(te), return-te: te)
<tuple-te>    ::= <non-empty-tuple-te> | <empty-te>
<non-empty-tuple-te> ::= ( <non-tuple-te> *)* <non-tuple-te> // tuple-te(tes: list(te))
<empty-te>    ::= Empty // empt-te()
<tvar>        ::= a symbol starting with T // tvar(id: Symbol)
```

2. תיוג הטיפוסים

שפות שונות מאפשרות להגדיר את הטיפוסים של הביטויים בתוכנית, באופנים שונים:

Java

```
int x = 3;
boolean f(int i, int j) { return i > j; }
```

TS

```
const x : number = 3;
const f : (i : number, j : number) => boolean = (i : number ,j : number) : boolean => i > j;
```

ב-5L נאמץ את הגישה של TS בצורה אלגנטית יותר:

```
(define [x : number] 3)
(define [f : [number * number -> boolean]] (lambda ([i :number]  [j : number]) [: boolean] (> i j)))
```

הגדרה פורמאלית של תחביר השפה 5L עם תיוג הטיפוסים (התוספות על 4L **מודגשות**):

```
<program> ::= (L5 <exp>+) // program(exps:List(exp))
<exp> ::= <define-exp> | <cexp>
<define-exp> ::= (define <var-decl> <cexp>) // def-exp(var:var-decl, val:cexp)
<cexp> ::= <num-exp> // num-exp(val:Number)
    | <bool-exp>  // bool-exp(val:Boolean)
    | <prim-op>   // prim-op(op:Symbol)
    | <var-ref>   // var-ref(var:Symbol)
    | (if <exp> <exp> <exp>) // if-exp(test,then,else)
```

```
| (quote <sexp>) // lit-exp(val:Sexp)
| (let (<binding>*) <cexp>+) // let-exp(bindings:List(binding), body:List(cexp))
| (letrec (<binding>*) <cexp>+) // letrec-exp(bindings:List(binding), body:List(cexp))
| (<cexp> <cexp>*) // app-exp(rator:cexp, rands:List(cexp))

| (lambda (<var-decl>*) [: <texp>]? <cexp>+)
    // proc-exp(params:List(var-decl), body:List(cexp), return-te: Texp) ##### L5
<var-decl>  ::= <symbol> | [<symbol> : <texp>] // var-decl(var:Symbol, type:Texp)  ##### L5
```

מימוש הפארסר

מערכת הטיפוסים: TExp.ts

תיוג הטיפוסים: T5-ast.ts

כמו הפארסר של T4, בתוספת קריאה לפארסר של הטיפוסים (אם הם לא מוגדרים מוגדר משתה טיפוס - T1, T2...): parseVarDecl, parseProcExp

3.2 סמנטיקה

הסמנטיקה קובעת כזכור את המשמעות של המבנים התחביריים. בפרק הקודם, הסמנטיקה קבעה את הערך של כל סוג ביטוי בשפה. בפרק זה, הסמנטיקה קובעת את המשמעות של טיפוסי הביטויים – האם טיפוסי הביטויים השונים בתוכנית תואמים זה לזה.

ה Type Checker מקבל ביטוי E בשפה L5 (כלומר AST בכולל בתוכו גם טיפוסים) ובודק האם הטיפוסים של כל תתי הביטויים ב-E תואמים זה לזה, ע"פ חוקי תאימות הטיפוסים. לבסוף הוא מחזיר את הטיפוס של הביטוי E.
[כזכור, בשפה עם טיפוסים, כמו 5L, ניתן לבצע בדיקה זו 'בזמן קומפילציה', על ה AST]

←

- נגדיר חוקי טיפוסים באופן פורמאלי
- נממש את החוקים ב type checker

1. הגדרה פורמאלית של חוקי טיפוסים

תשתית:

כאשר הגדרנו את חוקי החישוב עבור האינטרפרטר, השתמשנו בפרמטר של סביבה, כאשר סביבה זו הכילה את הערכים של משתנים שהוגדרו קודם. בחוקי הטיפוסים נשתמש גם כן בסביבה , הסביבה במקרה זה תכיל את הטיפוסים הידועים עד כה עבור משתנים שונים.

לדוגמא:

```
{  x : number, y : [number -> T5] }
```

באופן זה, הפרוצדורה apply_env מקבלת שם של משתנה ומחזירה את הטיפוס שלו ע"פ הסביבה.

בתיאור החוקים נשתמש במבנה הפורמאלי הבא (המכונה Typing Statement):

Tenv |-- e : t

שמשמעו: בהינתן הסביבה Tenv, מתקיים שהטיפוס של הביטוי e הוא t.

במקרה הכללי הצהרת טיפוס היא ביטוי בוליאני שערכו אמת או שקר. לדוגמא:
ערך הצהרת הטיפוס הבאה היא שקר

{ x : boolean } |-- (f x) : boolean

ניתן להסיק אמנם שהפרוצדורה f מקבלת פרמטר בוליאני, אך לא ניתן להסיק שגם הערך המוחזר הוא בולאני.

לעומת זאת, ערך הצהרת הטיפוס הבאה הוא אמת:

{ f : boolean -> boolean, x : boolean } |-- (f x) : boolean

שימו לב כי ערך ההצהרה הבאה הוא שקר:

{ f : boolean -> boolean } |-- (f x) : boolean

הסיבה – הפרמטר x אינו מוגדר בסביבת הטיפוסים, כך שייתכן f הופעלה עם x בעל טיפוס שגוי ובמקום להחזיר boolean יש שגיאת זמן ריצה.

← יש צורך להגדיר בסביבה הנחות על המשתנים החופשיים המופיעים בביטוי מימין.

{} |- (lambda (x) (+ x 3)) : [number -> number]

ערך ההצהרה הבאה הוא אמת: אין משתנים חופשיים בביטוי מימין, כך שלא נדרשת כל הנחת טיפוסים בסביבה. ע"פ הסמנטיקה של פעולת +, ניתן להסיק כי x הוא מספר.

הגדרת חוקי הטיפוס מבוססת על הצהרת טיפוס שהן אמת.

<u>חוקי הטיפוס</u>:

<u>Numbers</u>
Example: the type of ‘4’ is ‘number’
_Tenv |-- <num-exp _e> : number

<u>Booleans</u>
Example: the type of ‘#t’ is ‘boolean’
_Tenv |-- <bool-exp _e> : boolean

<u>Strings</u>
Example: the type of “abc” is ‘string’
_Tenv |-- <str-exp _e> : string

VarRef

Example: the type of ‘x’ given the environment { x: number} is ‘number’

_Tenv |-- <var-ref e> : apply_env(_Tenv,e)

Primitive operators

Example: the type of ‘+’ is [number * number -> number]

_Tenv |-- + : [number * number -> number]

Example: the type of ‘not’ is [boolean -> boolean]

_Tenv |-- not : [boolean -> boolean]

[בהגדרה זו של חוק הטיפוס ל not, אנו דורשים כי הפרמטר יהיה בוליאני. ניתן היה גם של לאלץ זאת, כלומר: [_Tenv |-- not : [_S -> boolean]

… [all remain primitive operators]

בניגוד לביטויים האטומיים עד כה, הביטויים להלן מורכבים, כך שמלבד הגדרת הטיפוס שלהם, חוק הטיפוס מתייחס גם לתאימות בין תתי הטיפוסים שלהם.

If

Example: the type of ‘(if (> x 5) y z)’ is…

- האם אנו רוצים לאלץ את החלק של ה test להיות boolean [האם אנו רוצים לתמוך בביטויים כמו (if 1 y z)]?

- האם אנו רוצים לאלץ שהטיפוס של ה then יהיה כמו הטיפוס של ה else [האם אנו רוצים לתמוך בביטויים כמו (if (3 > 4) #t 16)]?

בשפה כמו 5L, שאין בה איחוד טיפוסים, לא נוכל להצהיר על הטיפוס של מבנה ה if, כמו כן, מבחינה עיצובית זה לא כל כך אלגנטי להחזיר טיפוסים שונים בהתאם לתרחיש.

🡸 נאלץ את שתי הנקודות הנ"ל: התנאי חייב להיות בוליאני, טיפוסי ה then וה else חייבים להיות תואמים.

Given: <if-exp _e>

If _Tenv |-- _e.test : boolean
  _Tenv |-- _e.then : _S
  _Tenv |-- _e.alt : _S
Then _Tenv |-- _e : _S

Procedures

Example: the type of ‘(lambda (x : number) (* x x))’ is ‘number-> typeof(‘(* x x)’)

Given <proc-exp _e>

If _Tenv o { $\_e.params_1$.var : $\_e.params_1$.texp, …, $\_e.params_n$.var : $\_e.params_n$.texp } |--
  $\_e.body_1$: $\_S_1$, … ,$\_e.body_M$: $\_S_M$
  _e.returnTE : $\_S_M$
Then _e : [$\_e.params_1$.texp * … * $\_e.params_n$.texp -> $\_S_M$]

חוק זה למעשה קובע שטיפוס הערך המוחזר כפי שהוגדר בפרוצדורה חייב להיות תואם לטיפוס של הביטוי האחרון ב body.
לדוגמא, הביטוי '(lambda (x : number) : [boolean] (* x x))' לא יעבור type checking, מכיוון שטיפוס הערך המוחזר שהוגדר במפורש הוא boolean בעוד שטיפוס הביטוי האחרון ב body הוא number (כפי שמוגדר בחוק הטיפוס של *)

כלומר, הטיפוס של פרוצדורה נתונה הוא 'חתימה' בה טיפוסי הפרמטרים הם הטיפוסים שהוגדרו עבור הפרמטרים של הפרוצדורה, וטיפוס הערך המוחזר הוא הטיפוס של הביטוי האחרון בגוף הפרוצדורה.

<u>Application</u>
Example: (f 3)

Given: <app-exp _e>
If _Tenv |-- _e.rator : [$\_S_1$ * … * $\_S_n$ -> _S]
  _Tenv |-- $\_e.rands_1$: $\_S_1$, … $\_e.rands_n$: $\_S_n$
Then _e : _S

כלומר, החוק דורש תאימות של טיפוסי האופרנדים לפרמטרים של הפרוצדורה (ע"פ הגדרתם), ומגדיר את הטיפוס של ביטוי הפעלת הפרוצדורה להיות טיפוס הערך המוחזר של הפרוצדורה (ע"פ הגדרתה).

2. מימוש חוקי הטיפוסים

L5-typecheck.ts - הפרוצדורה typeOf

3.3 הסק טיפוסים (Type Inference)

עד כה, קיבלנו תוכנית עם טיפוסים, ובדקנו האם הם תואמים כפי שהוגדר בחוקים.
כעת, נרחיק לכת, וננסה להסיק בעצמנו את הטיפוסים שלא הוגדרו התוכנית.
כלומר, נקבל תוכנית שבה לא כל הטיפוסים בהכרח מוגדרים (אם הם לא מוגדרים, הפארסר מוסיף בשדה של הטיפוס 'משתנה טיפוס' (T12, T45,...)), ונחזיר תוכנית בה הטיפוסים הניתנים להסק מפוענחים.

3.3.1 דוגמאות

א.

```
(
 (lambda (x) (+ x 3))
 5
)
```

מה הטיפוס של x?
מהו טיפוס הערך המוחזר של הפרוצדורה?

ניתן לראות ש x הוא number: טיפוס האופרנד שנשלח עבורו, 5, הוא מספר; x עצמו נשלח כאופרנד לאופרטור +, המצפה לקבל פרמטרים מסוג number.
ניתן לראות שחתימת הפרוצדורה היא: number -> number: טיפוס הפרמטר (x) הוא number, טיפוס גוף הפרוצדורה הוא מספר (ע"פ הגדרת אופרטור +)

<u>ננסה להכניס תובנות אלו תחת מסגרת שיטתית</u>

- נסמן כל תת ביטוי ע"י משתנה טיפוס

((lambda (x) (+ x 3)) 5) : T1
(lambda (x) (+ x 3)) : T2
(+ x 3) : T3
+ : T4
x : T5
3 : T6
5 : T7

- נתאר את כל התובנות על טיפוסי תתי הביטויים כמשוואות

T4 = number * number -> number (חתימת אופרטור פרימיטיבי)
T5 = T7 (הצבת אופרנד לפרמטר של פרוצדורה)
T2 = T5 -> T3 (הגדרת טיפוס פרוצדורה)
T7 = number (חוק הטיפוס עבור ביטוי מספרי)
T3 = number (האופרטור הפרימיטיבי + מחזיר מספר)

- נפתור את מערכת המשוואות

T5 = number (טיפוס המשתנה איקס)
T2 = number -> number (טיפוס הפרוצדורה)

(
(lambda ([**x : number**]) [**: number**] (+ x 3))
5
)

ב.

(lambda (x) (x x))

מה הטיפוס של המשתנה x?
מה טיפוס הערך המוחזר של הפרוצדורה?

- נציין את הטיפוס של כל תת ביטוי על ידי משתנה טיפוס

(lambda (x) (x x)) : T1
(x x) : T2

x : T3

- נתאר את התובנות השונות על טיפוסי תתי הביטויים כמשוואות

T1 = T3 -> T2 (הגדרת פרוצדורה)

T3 = T3 -> T2 (הטיפוס של הפרוצדורה איקס על פי ההפעלה שלה)

T3 = T3 (משוואת זהות)

Contradiction

הסתירה בפתרון המשוואות מלמדת שלא ניתן להסיק את הטיפוס. יתרה מזאת, אין הצבה של טיפוסים קונסיסטנטית עבור תוכנית זאת.

ג.

(lambda (f x) (f x x))

מה הטיפוס של המשתנה f?
מה הטיפוס של המשתנה x?
מה טיפוס הערך המוחזר של הפרוצדורה?

- נציין את הטיפוס של כל תת ביטוי על ידי משתנה טיפוס

(lambda (f x) (f x x)) : T1
(f x x) : T2
f : T3
x : T4

- נתאר את תובנות הטיפוסים כמשוואות

T3 = T4 * T4 -> T2 (סמנטיקת הפעלת פרוצדורה)
T1 = T3 * T4 -> T2 (סמנטיקת הגדרת פרוצדורה)

- נפתור את המשוואות

T1 = (T4 * T4 -> T2) * T4 - > T2

פולימורפי, יכול לעבוד עם זוגות שונים של T2,T4 המקיימים את פתרון המשוואה.

3.3.2 אלגוריתם פורמאלי להסק טיפוסים

נתון ביטוי e, יש למצוא את הטיפוס של כל ה VarDecl ושל הערך המוחזר עבור הפרוצדורות (תכלס, את כל הטיפוסים הניתנים להסקה).

שלבי האלגוריתם באופן כללי:

1. [שינוי שמות המשתנים בתוכנית לשמות יחודיים (האלגוריתם ישתמש בסביבה (שלב 3 להלן), כדי להימנע מתחזוק היררכיה של פריימים אנחנו מראש קובעים שאין שני משתנים בעלי אותו שם)]
2. הגדרת משתנה טיפוס לכל תת ביטוי
3. יצירת מערכת משוואות המתארות את קשרי הטיפוסים בין משתני הטיפוסים
4. פתרון מערכת המשוואות

<u>דוגמא מלווה</u>:

```
(lambda (f g)
  (lambda (x)
    (f (+ x (g 3)))))
```

4.3.2.1 מתן שמות יחודיים לכל משתנה

בדוגמא שלנו לכל המשתנים שמות יחודיים, אין צורך לשנות את שמותם

4.3.2.2 הגדרת משתנה טיפוס לכל תת ביטוי

(lambda (f g) (lambda (x) (f (+ x (g 3)))))	T0
(lambda (x) (f (+ x (g 3))))	T1
(f (+ x (g 3)))	T2
f	Tf
(+ x (g 3))	T3
+	T+
x	Tx
(g 3)	T4
g	Tg
3	Tnum3

<u>קוד</u>: הפונקציה expToPool בקובץ L5-type-equations.ts

4.3.2.3 יצירת משוואות הטיפוס

- ביטויים אטומיים

משווה ע"פ הטיפוס המוגדר

T_{num} = number
T_{bool} = boolean
...
T+ = number * number -> number

- If

משוואות ע"פ האילוצים

T_{test} = boolean
$T_{then} = T_{alt}$

- הגדרת פרוצדורה

משוואות ע"פ האילוצים

נתון: (lambda (p_1...p_n) e_1...e_m)

T_{lam} = [Tp_1 *... * Tp_n -> Te_m]

[אם אין פרמטרים T_{lam} = Empty -> Te_m]

- הפעלת פרוצדורה

משוואות ע"פ האילוצים

נתון: (f a_1...a_n)

Tf = [Ta_1 *... * Ta_n -> Tapp]

בדוגמא שלנו

Expression ,Tvar	**Equation**
3 , Tnum3	Tnum3 = Number
+ , T+	T+ = Nnumber * Number -> Number
(lambda (f g) (lambda (x) (f (+ x (g 3))))) , T0	T0 = [Tf * Tg -> T1]
(lambda (x) (f (+ x (g 3)))) , T1	T1 = [Tx -> T2]
(f (+ x (g 3))) , T2	Tf = [T3 -> T2]
(+ x (g 3)) , T3	T+ = [Tx * T4 -> T3]
(g 3) , T4	Tg = [Tnum3 -> T4]

<u>קוד</u>: הפונקציה makeEquationsFromExp בקובץ L5-type-equations.ts

4.3.2.4 פתרון מערכת משוואות הטיפוס

<u>תשתית</u>: הצבת טיפוסים והפעולות עליה

- הצבת טיפוסים (Type Substitution)

מבנה נתונים הממפה <u>משתני טיפוס</u> (T1, T3) ל<u>ביטויי טיפוס</u> (number, boolean -> T5)
[מבנה זה דומה לסביבה בה השתמשנו ב type-checker, אך שם מופו <u>משתנים בתוכנית</u> (x,y) לביטויי טיפוס, וכאן אנו ממפים <u>משתני טיפוס</u> (T1, T5) לביטויי טיפוס שכאלה]

לדוגמא:

{ T1 = number, T2 = [[number -> T3] -> T4] }

אילוץ: משתנה הטיפוס משמאל אינו יכול להופיע בביטוי הטיפוס מימין.
לדוגמא, T2 = number -> T2 אינו חוקי.

אלגוריתם פתרון המשוואות יתחיל עם הצבה ריקה, שתלך ותגדל כל פעם נגיע לתובנות חדשות על משתני הטיפוס בזכות המשוואות.

- הפעלת הצבה (Substitution application)

הפעלת הצבה על ביטוי 'מפרשת' אותו ע"י הצבת משתני הטיפוס המוגדרים בהצבה במופעים שלהם בביטוי. כלומר, החלפת כל משתנה טיפוס בביטוי ב RHS של משתנה זה בהצבה (אם הוא קיים שם).

לדוגמא:

[[T1->T2]->T2] o {T1=Boolean, T2=[T3->T3]} = [[Boolean->[T3->T3]] -> [T3->T3]]

- הרכבת הצבות (Substitution combination)

בהינתן שתי הצבות, פעולת ההרכבה תהפוך אותם להצבה אחת כוללת.

<u>המדיניות הכללית</u>: ניקח כבסיס את אחת ההצבות (ההצבה הראשונה), ונשתמש בהצבה השניה כדי להעשיר אותה (לפרש משתני טיפוס בהצבה הראשונה, להוסיף הגדרות של משתני טיפוס חדשים), תוך סינון הגדרות משתנים הסותרות את המוגדר בהצבה הראשונה.

בפועל, בהינתן שתי הצבות 'S,S, נגדיר את פעולת ההרכבה 'S o S באופן הבא:

- החלפת משתני טיפוס המופיעים בביטויי הטיפוס (ב-RHS) ב S, בהגדרתם (אם קיימת) ב-'S. במילים אחרות, פרשנות משתני טיפוס ב S ע"פ הידוע עליהם ב 'S.
- הוספת הגדרות של משתני טיפוס ב-'S ל S, אם ה אינם מוגדרים עדיין ב-S (כלומר, מסננים הגדרות משתני טיפוס ב 'S הסותרים את הגדרם ב-S)

דוגמא:

{T1=Number, T2=[[Number->T3] -> T5]} o {T3=Boolean, T1=[T2->T2], T4 = Number} =
{ T1 = Number, T2 = [[Number-> Boolean] -> T5], T3 = Boolean, T4 = Number }

קוד: הקובץ <u>L5-substitution-adt.ts</u>, בפרט הפרוצדורות applySub, combineSub

<u>הערה</u>: יוניפיקציה (Unification)

בעזרת קונספט זה של הצבה, ניתן להגדיר את יחס הספציפיות בין טיפוסים.
כזכור, הגדרנו יחס זה כבר בפרק הראשון, כיחס הכלה בין קבוצות (חיות - כלבים). ניתן להגדיר אותו גם ע"י הפעלת הצבה באופן הבא:
נאמר כי טיפוס 'T ספציפי מטיפוס T, אם קיימת הצבה S כך ש T o S = T' (כלומר, 'T הוא פרשנות ספציפית של הטיפוס הכללי יותר T)

לדוגמא:

T = [T1 -> T1]
T' = [number -> number]

'T ספציפית מ T מבחינת קריטריון הכלת הקבוצות.
'T ספציפית מ T גם מבחינת הקריטריון החדש, כי 'T הוא פירוש ספציפי של T ע"פ ההצבה { T1 = number} :

T o { T1 = number } = T'

נאמר כי שני ביטויי טיפוס נחשבים ל<u>מאוחדים</u> אם קיימת הצבה ההופכת אותם לזהים (כלומר, קיים עולם שבו יש להם פירוש זהה). הצבה זו מכונה **unifier**

T o S = T' o S

לדוגמא:

T = [T4 * [number -> T1] -> T4]
T’ = [Pair(T2) * [T2 -> T2] - > T3]
S = { T4 = Pair(number), T2 = number, T1 = number, T3 = Pair(Number)}

T o S = T’ o S = [Pair(number) * [number -> number] -> Pair(number)]

בהינתן שני ביטויי טיפוס, נגדיר את המאחד הכללי ביותר שלהם – Most General Unifier (MGU) – כהצבה הכללית ביותר מבין כל ההצבות המאחדות בין שני ביטויי הטיפוס.

לדוגמא:

T = [T3 * T3 -> T3]
T’ = [Pair(T1) * T2 -> T2]

ניתן לאחד את T ו 'T ע"י ההצבה S וכן ע"י ההצבה 'S

S = { T3 = Pair(T1}, T2 = Pair(T1) }
S’ = { T3 = Pair(number}, T2 = Pair(number) , T1 = number}

ההצבה S היא ה MGU כי היא כללית יותר מ ’S.

האלגוריתם להלן מוצא את ה MGU של משתני הטיפוס שיצרנו עבור הביטויים בתוכנית. כלומר, האלגוריתם מוצא את ההצבה הכללית ביותר עבורם המקיימת את המשוואות שיוצרו בשלב הקודם.

<u>האלגוריתם</u>:

אתחול: מאגר משוואות, הצבה ריקה

לולאה:

כל עוד יש משוואות במאגר:

- הסרת משוואה אחת מהמאגר, והפעלת ההצבה על שני צדדיה (=פרשנות המשוואה ע"פ הידע שהצטבר עד כה על משתני הטיפוס)

- טיפול במשוואה ע"פ שלושה מקרים:

  - שני הצדדים אטומיים והם אינם משתני טיפוס (number=number, number=boolean):
    בדיקה שהם זהים, אחרת שגיאה

  - אחד הצדדים הוא משתנה טיפוס (T1=number, boolean = T4, T2=T3,):
    הרכבת ההצבה הנוכחית עם הצבה הכוללת את המשוואה הנתונה (הוספת התובנה ש T1=number וש T2=T3 להצבה)

- שני הביטויים במשוואה מורכבים (טיפוסי פרוצדורה בשפה שלנו) ובעלי אותו מבנה:
  (T1->boolean = number->T2)
  פירוק רכיבי הביטויים במשוואה והוספת המשוואות המקבילות להצבה
  (T1 = number, T2 = boolean)

אחרת: שגיאה

בדוגמא שלנו

Equations	substitution
1. T0 = [Tf * Tg -> T1]	{}
2: T1 = [Tx -> T2]	
3: Tf = [T3 -> T2]	
4: T+ = [Tx * T4 -> T3]	
5: Tg = [Tnum3 -> T4]	
6: Tnum3 = Number	
7: T+ = Number * Number -> Number	

Equation 1 is processed by the case 2 of the algorithm - since one of its sides is a type variable (T0).

Equations	substitution
2: T1 = [Tx -> T2]	{**T0 = [Tf * Tg -> T1]**}
3: Tf = [T3 -> T2]	
4: T+ = [Tx * T4 -> T3]	
5: Tg = [Tnum3 -> T4]	
6: Tnum3 = Number	
7: T+ = Number * Number -> Number	

In the second iteration, we process the second equation, and again apply case 2 of the algorithm:
Note how the substitution composition resulted in the transformation of the T1 argument by its value in the right hand side of T0 in the substitution.

Equations	substitution
3: Tf = [T3 -> T2]	{T0 = [Tf * Tg -> **[Tx -> T2]**, **T1 = [Tx -> T2]**}
4: T+ = [Tx * T4 -> T3]	
5: Tg = [Tnum3 -> T4]	
6: Tnum3 = Number	
7: T+ = Number * Number -> Number	

Same case 2 for the third equation.

Equations	substitution
4: T+ = [Tx * T4 -> T3]	{T0 = [**[T3 -> T2]** * Tg -> [Tx -> T2]], T1 = [Tx -> T2], **Tf = [T3 -> T2]**}
5: Tg = [Tnum3 -> T4]	
6: Tnum3 = Number	
7: T+ = Number * Number -> Number	

Again case 2 for the 4th equation:

Equations	substitution
5: Tg = [Tnum3 -> T4]	{T0 = [[T3 -> T2] * Tg -> [Tx -> T2]], T1 = [Tx -> T2], Tf = [T3 -> T2],
6: Tnum3 = Number	**T+ = [Tx * T4 -> T3]**}
7: T+ = Number * Number -> Number	

One more case 2 for the 5th equation:

Equations	substitution
6: Tnum3 = Number	{T0 = [[T3 -> T2] * **[Tnum3 -> T4]** -> [Tx -> T2]], T1 = [Tx -> T2],
7: T+ = Number * Number -> Number	Tf = [T3 -> T2], T+ = [Tx * T4 -> T3], **Tg = [Tnum3 -> T4]**}

One more case 2 for the 6th equation.
Note how the substitution composition resulted in the transformation of the Tnum argument by its value in the right hand side of Tg in the substitution.

Equations	substitution
7: T+ = Number * Number -> Number	{T0 = [[T3 -> T2] * [**Number** -> T4] -> [Tx -> T2]],
	T1 = [Tx -> T2], Tf = [T3 -> T2],
	T+ = [Tx * T4 -> T3],
	Tg = [**Number** -> T4],
	Tnum3 = Number }

Equation 7 is first interpreted by the substitution as: **[Tx * T4 -> T3]** = [Number * Number -> Number]

This is case 3 of the algorithm - we split the two sides of the equation into components - because they have the same type constructor (proc-te) and yield the new equations:

Tx = Number
T4 = Number
T3 = Number

Equations	substitution
8: Tx = Number	{T0 = [[T3 -> T2] * [Number -> T4] -> [Tx -> T2]],
9: T4 = Number	T1 = [Tx -> T2], Tf = [T3 -> T2],
10: T3 = Number	T+ = [Tx * T4 -> T3],
	Tg = [Number -> T4],
	Tnum3 **=** Number }

Case 2 for equation 8:

Equations	substitution
9: T4 = Number	{T0 = [[T3 -> T2] * [Number -> T4] -> [**Number** -> T2]],
10: T3 = Number	T1 = [**Number** -> T2], Tf = [T3 -> T2],
	T+ = [**Number** * T4 -> T3],
	Tg = [Number -> T4],
	Tnum3 **=** Number, **Tx = Number** }

Case 2 for equation 9:

Equations	substitution
10: T3 = Number	{T0 = [[T3 -> T2] * [Number -> **Number**] -> [Number -> T2]], T1 = [Number -> T2], Tf = [T3 -> T2], T+ = [Number * **Number** -> T3], Tg = [Number -> **Number**], Tnum3 = Number, Tx = Number, **T4 = Number** }

Case 2 for equation 9:

Equations	substitution
	{T0 = [[**Number** -> T2] * [Number -> Number] -> [Number -> T2]], T1 = [Number -> T2], Tf = [Number -> T2], T+ = [Number * Number -> **Number**], Tg = [Number -> Number], Tnum3 = Number, Tx = Number, T4 = Number, **T3 = Number** }

We eventually output the following substitution:

{T0 = [[Number -> T2] * [Number -> Number] -> [Number -> T2]],
T1 = [Number -> T2],
Tf = [Number -> T2],
T+ = [Number * Number -> Number],
Tg = [Number -> Number],
Tnum3 = Number,
Tx = Number
T4 = Number
T3 = Number}

On the basis of this substitution, we can return the fully annotated expression:

(lambda (**[f : [Number -> T2] [g : [Number -> Number]]**) **: [Number -> T2]**
(lambda (**[x : Number]**) **: T2**
(f (+ x (g 3)))))

קוד: הפונקציה solveEquations בקובץ L5-type-equations.ts

התכנסות אלגוריתם ההסק

טענה: אלגוריתם ההסק מתכנס

הוכחה: נסמן את 'מצב' האלגוריתם ע"י הזוג <D,N>, כאשר N הוא מספר המשוואות הנוכחי, ו D הוא עומק ביטוי הטיפוס המורכב המופיע במשוואות.
בכל שלב בלולאה, או ש N קטן באחד (מקרים 1,2), או ש D קטן (מקרה 3, מוריד את רמת המורכבות של אחת המשוואות). כלומר בסוף N=0 או D=0, כלומר אין יותר משוואות.

מימוש קומפקטי ויעיל יותר של האלגוריתם

<u>אבחנות</u>:

- כל תת-ביטוי שחילצנו ובחרנו עבורו משתנה טיפוס חדש בשלב 1 של אלגוריתם ההסק, הוא למעשה קודקוד בעץ התחבירי המופשט (ה-AST) של הביטוי הנתון. כלומר, כל אחד מתתי-הביטויים נבחן על ידי ה type checker. יתרה מכך, הפארסר מייצר משתנה טיפוס לכל המשתנים המוגדרים ולערך המוחזר של הפרוצדורות (כלומר, לכל הטיפוסים שאנחנו רוצים להסיק מה הם)
- משוואות הטיפוס שגזרנו בשלב 2 של אלגוריתם ההסק, ע"פ תתי הביטויים השונים, תואמות אחת לאחת לאילוצים שבדקנו ב Type Checker עבור כל קודקוד ב AST.

🡸 נממש את כל אלגוריתם ההסק כחלק מה Type Checker. כלומר נריץ את ה Type Checker ללא כל הסק מקדים, תוך עדכון קל – מימוש שלב 3 של אלגוריתם ההסק (פתרון המשוואות) כחלק מה type checking: כאשר נבדקת התאימות בין שני טיפוסים – הפרוצדורה checkEqualType – אם אחד הטיפוסים הוא משתנה טיפוס (T12), במקום להחזיר שגיאה אם אין תאימות (מצד אחד T12 מצד שני number), נציין כי הטיפוס של משתנה הטיפוס הוא ביטוי הטיפוס.

לדוגמא:

```
(
 (lambda (x) x)
 3
)
```

הפארסר יצמיד למשתנה x בפרוצדורה משתנה טיפוס חדש, נניח T1.
לאחר מכן, ה type-checker במסגרת המעבר עם תתי הביטויים, יגיע לתרחיש של AppExp ויבדוק האם הטיפוס של האופרנד 3 תואם לטיפוס של הפרמטר x. הטיפוס של 3 הוא number, הטיפוס של x הוא T1 – כך שה type-checker יחזיר Failure.
אם נשנה את checkEqualType כך שעבור השוואת שני טיפוסים שאחד מהם הוא משתנה טיפוס (T1) באופן כזה שהיא לא תחזיר false אלא תציין שמשתנה הטיפוס (T1) הוא מכאן ואילך הטיפוס שאליו הוא הושווה (number)

בהמשך, אם ניתקל שוב בבדיקת הטיפוס של משתנה זה, נצרף שוב את הטיפוס שאליו הוא מושווה לרשימות הטיפוסים האפשריים עבורו:

```
(define f (lambda (x) x))

...

(f 2)
(f y)
```

בכל קריאה ל f בוצעה בדיקה של תאימות טיפוס האופרנד לטיפוס הפרמטר.
אם הטיפוס של x לאחר הפארסינג הוא נניח T1, בבדיקת הטיפוסים של הקריאה הראשונה שדה התוכן שלו יהיה number. בבדיקת הטיפוס בקריאה השניה, שדה התוכן שלו יתעדכן גם עם הטיפוס של y (נניח משתנה הטיפוס T2) 🡸 שדה התוכן אוגר לתוכו את כל ההצבות עבור משתנה הטיפוס, כך שההרכבה שלהם היא למעשה 'פתרון' המשוואות עבורו. בפרט T1=T2=number

במידה שתהיה סתירה בשדה התוכן. לדוגמא:

```
(f 2)
(f y)
(f #t)
```

כך שייווצר שדה תוכן: T1=T2=number=boolean
נסיק שיש בעיה בטיפוסי התוכנית / לא ניתן להסיקם (בשפה שלנו בה אין איחוד טיפוסים, אם יש איחוד שכזה בתחביר, ניתן יהיה להסיק במקרה זה של T1=T2=number|boolean)

מימוש: הפרוצדורה checkEqualType בקובץ L5-typeinference.ts

5. מבני בקרה

מבני בקרה בשפות תכנות קובעים את סדר ביצוע הפקודות.
עד כה התוודענו למבני בקרה שונים:

- הרצה סדרתית של פקודות
- מבנה if-else
- לולאה
- קריאה לפרוצדורה
- זריקת exception

בפרק זה, נבחן שני מבני בקרה נוספים, המאפשרים לנו, כמתכנתים בשפת ה-L, לשלוט על זמן החישוב של הביטויים. בפרט, להשהות את החישוב של ביטויים שונים בתוכנית לזמן מאוחר יותר:

- קריאות אסינכרוניות
- קו-רוטינות

נדגים תחילה מבנים אלו בשימוש מודרני ואופנתי ב JavaScript, ולאחר מכן נממש אותם בשפת L.

במסגרת זאת, נעדכן גם את האינטרפרטר – L6,L7 (לא מדובר בשפות חדשות, אין סוגים חדשים של ביטויים, אלא באינטרפרטר יעיל יותר)

4.1 גישה אסינכרונית ל i/o ב JavaScript

נתקלנו כבר בעבר במנגנונים בהם ביטויים/פקודות מסויימים מחושבים/מבוצעות בזמן מאוחר יותר:

- יצירת ת'רד

```
Thread t = new Thread(() => { … });
t.start();
```

- טיפול בבקשת לקוח בשרת הריאקטור

בשני מקרים אלו, הקוד לא בוצע באופן סנכרוני, כלומר ביצוע כעת של המשימה שורה אחרי שורה, אלא באופן אסינכרוני, כלומר, הגדרנו משימה (ת'רד, טיפול בלקוח), העברנו אותה למנגנון שייטפל בה בהמשך (סביבת הריצה תריץ את הת'רד, השרת יגיב בהמשך ל i/o זמין של הלקוח), והמשכנו הלאה לשורת הקוד הבאה על אף שהמשימה לא בוצעה עדיין.

שתי דוגמאות נוספות למנגנונים שכאלה:

- ה event loop של ה browser (גוגל כרום, אקספלורר)
  - ניתוח דף ה html
  - הרצת סקריפטים של JS המוגדרים בדף
  - תגובות לפעולות משתמש
  - תגובות להודעות מהשרת

לדוגמא:

אם מופיע בדף ה html:

setTimeout(()=> { console.log('abc') }, 1000)
undefined

abc

$("btn_1").click(() => alert("Btn 1 clicked"));

- ה event loop של node (סביבת הריצה של JS)

הגישה ל i/o לוקחת זמן רב (בסדר גודל עצום ביחס לפעולות בזיכרון). קריאה סנכרונית תמתין עד שהפעולה תתבצע. בקריאה אסינכרונית הבקשה מועברת למערכת ההפעלה עם פונקציית callback לביצוע כאשר הגישה ל i/o תסתיים, והתוכנית ממשיכה. כאשר מערכת ההפעלה תסיים את הקריאה/כתיבה/... מ/ל io תתווסף משימה של ביצוע ה callback ל eventloop של node.

כפי שנאמר כבר, ניתן לגשת למידע בקובץ (לקריאה או לכתיבה) באופן **סינכרוני**, כך שהפעולה לא תסתיים עד אשר ניגש למידע, או לחלופין באופן **א-סינכרוני**, כך שהפעולה הנוכחית מסתיימת מייד והטיפול במידע יתרחש במועד מאוחר יותר, לאחר שהמידע יהיה נגיש.

<u>גישה סנכרונית לקבצים</u>

קריאה מקובץ

```
// Synchronous (blocking) call to readFileSync
// The return value of the readFileSync procedure can be passed directly
to the JSON.parse function.
const readJSONSync = (filename) => {
  return JSON.parse(fs.readFileSync(filename, 'utf8'));
}
```

תחילה, אנו מבקשים ממערכת ההפעלה לקרוא באופן סנכרוני את תוכן הקובץ filename

fs.readFileSync(filename, 'utf8')

התוכנית תחכה עד אשר המידע ייקרא מהקובץ, ולאחר מכן, אנו מבצעים ניתוח של המחרוזת החוזרת לאובייקט Json

JSON.parse(fs.readFileSync(filename, 'utf8'**)**

כתיבה לקובץ

```
const writeJSONSync = (filename, map) => {
  return fs.writeFileSync(filename, JSON.stringfy(map), 'utf8');
}

writeJSONSync("test", {id:1, text:'hello'});
console.log(readJSONSync("test"));
```

אנו מבקשים ממערכת ההפעלה לכתוב באופן סנכרוני מחרוזת (המרה של אובייקט JSON) לקובץ filename

<u>גישה לא סנכרונית</u>

```
const readJSON = (filename) => {
  fs.readFile(filename, 'utf8', (err, res) => {
    if (err)
      console.log(err);
    else
      console.log(JSON.parse(res));
  });
}
```

בפעולת readFile אנו מבקשים ממערכת ההפעלה לקרוא באופן לא סינכרוני את תוכן הקובץ, ומבקשים מ node לבצע את הקלוז'ר המצורף על תוצאת הקריאה האסינכרונית (הפרמטרים res, err) לאחר שמערכת ההפעלה תסיים בהמשך את הקריאה.

כתיבה לקובץ

```
const writeJSON = (filename, map) => {
  fs.writeFile(filename, JSON.stringify(map), (err) => {
    if (err)
      console.error(err);
    else
      console.log("The file was saved: ", filename);
  });
  console.log("This is invoked before the callback is invoked.");
```

```
}
```

בגרסה זו אנו רק מבקשים ממערכת ההפעלה לכתוב את ייצוג המחרוזת של אובייקט ה JSON לקובץ filename, ומספקים את התגובה לפעולה זו בהמשך ע"י פונקציית ה callback (במקרה זה היא מקבלת רק פרטמר אחד, err, כי אין ערך מוחזר לפעולת הכתיבה (בניגוד לפעולת הקריאה שהחזירה את תוכן הקובץ)).

<u>הערה</u>: פונקציית ה callback שאנו מספקים היא למעשה closure הכולל לא רק את הקוד של הפונקציה אלא גם את 'הסביבה' שהיתה שבזמן היווצרותה.

לדוגמא

```
const writeJSON = (filename, map) => {

  let date = Date();

  fs.writeFile(filename, JSON.stringify(map), (err) => {
    if (err)
      console.error(err);
    else
      console.log(date + ": The file was saved: ", filename);
  });
  console.log("This is invoked before the callback is invoked.");
}
```

הקלוז'ר של ה callback שהגדרנו עבור פעולת הכתיבה, כולל לא רק את הקוד של הפונקציה אלא גם את המשתנה date שהוגדר מחוצה לה. כך שהתאריך שיודפס, יהיה התאריך בזמן הקריאה לפונקציה writeFile, ולא התאריך של זמן ביצוע ההדפסה.

<u>הרכבת קריאות אסינכרוניות</u>

לעתים נדרש לבצע שרשרת של פעולות על מידע.

לדוגמא, קריאת קובץ, עדכון מידע, וכתיבה של התוכן המעודכן לקובץ.

בקריאה סנכרונית, הדבר פשוט:

writeJsonSync(update(readJsonSync('a.json')), 'a.json')

בקריאה אסינכרונית זה מורכב יותר, שהרי הקריאה האסינכרונית הינה void. המידע ייקרא רק בהמשך, וייתכנו בהמשך שגיאות שצריך לטפל בהן.

← שרשור הפעולות יהיה למעשה שרשור של התגובות לכל פעולה, כלומר שרשור של callbacks.

לדוגמא: שרשור הפעולות הסינכרוניות (f(g(h(x))), ייראה בגרסה האסינכרונית כך:

```
h(x, (hRes) => {
  g(hRes, (gRes) => {
```

```
    f(gRes, (fRes)=> fRes);
  })
});
```

ועבור הדוגמא שלנו:

```
readJson('a.json', (err, res) => {
  if (err) …
  else
    writeJson('a.json', update(res), (err) => {…});
}
```

```
const readJSON = (filename, callback) => {
  fs.readFile(filename, 'utf8', calback);}
const writeJSON = (filename, map, callback) => {
  fs.writeFile(filename, JSON.stringify(map), callback);}
```

מסורבל משהו...
← נשתמש בעיצוב נוח להרכבת פונקציות אסינכרוניות: Promise

<u>Promise</u>

Promise הינו אובייקט עם השדות הבאים:
task – המשימה לביצוע
value – התוצאה של ביצוע המשימה
err – מאחסן את אפיון השגיאה של ביצוע המשימה (אם נכשלה)
state – מצבו הנוכחי של הפרומיס

Pending: המשימה לא התבצעה עדיין
Fulfilled: המשימה בוצעה והסתיימה בהצלחה
Rejected: המשימה בוצעה אך נכשלה

handlers – רשימת פונקציות לביצוע על תוצאת הפעלת המשימה (התגובות השונות לתוצאה)
then: הפונקציות לביצוע במקרה של הצלחה
catch: הפונקציות לביצוע במקרה של כישלון

אורח חייו של Promise:

- נוצר במצב Pending
- מתווסף לתור המשימות של הסביבה (node)
- כשמגיע תורו להתבצע (במסגרת ה event loop), מופעלת פונקציית ה task שלו
  - אם הצליחה: מעבר למצב fulfilled, יופעלו על התוצאה (השדה value) כל ההנדלרים של ההצלחה (הרשימה then)

- אם נכשלה: מעבר למצב rejected, יופעלו על השגיאה (השדה err) כל ההנדלרים של הכישלון (הרשימה catch)

נעצב מחדש את פעולות הקריאה והכתיבה בעזרת ה Promise:

<u>קריאה</u>

```
const readFilePromise = (filename: string) : Promise<string> => {
 return new Promise<string>( (resolve, reject) => {
  fs.readFile(filename, (err, res) => {
   if (err)
    reject(err);
   else
    resolve(res.toString('utf8'));
  })
 })
}
```

הבנאי מקבל כפרמטר את תיאור המשימה.

המשימה מוגדרת כפונקציה המקבלת שתי פונקציות, האחת הנדלר לטיפול בהצלחה (הפרמטר resolve), והשניה הנדלר לטיפול בכישלון (הפרמטר reject).
במקרה שלנו, קוד המשימה הוא קריאה מקובץ (באופן אסינכרוני, עם פונקציית callback), תוך הפעלת שני ההנדלרים על תוצאת פעולת הקריאה.

נשים לב לכך,כי בשלב זה, עדיין לא הוגדרו ההנדלרים (נעשה זאת מיד). כך שאם המשימה תבוצע לפני כן לא יהיו עדיין הפרמטרים resolve, reject. התוצאות של המשימה יישמרו בשדות value, err, ובכל פעם שיוגדר handler הוא יופעל על התוצאה, בהתאם להגדרת המשימה.

דוגמא לשימוש ב Promise:

```
const testContent : Promise<string> = readFilePromise('test.json');
testContent
    .then((content: string) => console.log("Content: ",
JSON.parse(content)))
    .catch((err) => console.error(err));
```

לאחר הגדרת ה Promise אנו מגדירים שני הנדלרים לטיפול בתוצאת הצלחה ולטיפול בכישלון (כמו בדוגמא המקורית).

<u>כתיבה לקובץ</u>

```
const writeFilePromise = (filename: string, content: string): Promise<void> => {
 return new Promise( (resolve, reject) => {
```

```
  fs.writeFile(filename, content, (err) => {
   if (err)
    reject(err);
   else
    resolve();
  })
 })
}
```

כעת ניתן להרכיב את שתי הפעולות האסינכרוניות, על ידי שרשור של ההנדלרים:

```
const readUpdateWrite = (filename: string) : Promise<void> => {
    return readFilePromise(filename)
            .then((content) => {
                let j = JSON.parse(content);
                j.lastModified = new Date(); // update
                return writeFilePromise(filename, JSON.stringify(j));
            })
            .catch((err) => console.error(err));
}
```

דוגמא להפעלה של הפעולה המורכבת:

```
writeFilePromise('test.json', JSON.stringify({a: 1}))
    .then(() => console.log("File is created"))
    .then(() => readFilePromise('test.json'))
    .then((content) => console.log(JSON.parse(content)))
    .then(() => readUpdateWrite('test.json))
    .then(() => console.log('File is updated'))
    .then(() => readFilePromise('test.json))
    .then((content) => console.log(JSON.parse(content)));
```

4.2 קו-רוטינות ב JavaScript, מחוללים (Generators)

קו-רוטינה (co-routine) היא פונקציה שניתן לבצע אותה בשלבים. כלומר, במהלך ביצוע גוף הפונקציה ניתן לחזור באמצע (עם תוצאת ביניים) לשורת הקריאה לפונקציה, ולחזור אליה בהמשך, לאותה נקודה, בקריאה חוזרת.

ב TypeScript קו-רוטינות מעוצבות בעזרת הממשקים הבאים:

```
interface Iterator {
  next() : IteratorResult;
}

interface IteratorResult {
   value : any;
   done : boolean;
}
```

הממשק Iterator מגדיר ביצוע של שלב אחד בפרוצדורה. והממשק IteratorResult מגדיר תוצאה של ביצוע שלב אחד בפרוצדורה (הערך value) כולל אינדיקציה האם זהו השלב האחרון (הערך done)

באופן זה, אובייקט הממשש את הממשק Iterator הינה למעשה קו-רוטינה, כאשר ביצוע כל שלב ניתן על ידי קריאה ל'מתודה' next, עד אשר מקבלים ערך אמת בשדה done.

ב JavaScript קיים תחביר נוח, syntactic sugar, להגדרת קו-רוטינות. לדוגמא:

```
function* idMaker () {
    yield 1;
    yield 2;
    yield 3;
}
```

מבנה זה הינו למעשה בנאי שמחזיר אובייקט מטיפוס Iterator, שכל פעולתת next שלו מריצה את הקוד עד ה yield הבא, עם תוצאת ביניים בעלת הערך שניתן ל yield. ה yield האחרון יחזיר ערך אמת עבור השדה done.

דוגמא לשימוש בקו-רוטינה הנ"ל:

```
let cr : Iterator = idMaker();
let ir : IteratorResult = cr.next();
console.log(ir.value); // 1
console.log(ir.done); // false
ir = cr.next();
console.log(ir.value); // 2
console.log(ir.done); // false
ir = cr.next();
console.log(ir.value); // 3
console.log(ir.done); // false
ir = cr.next();
console.log(ir.value); // undefined
console.log(ir.done); // true
```

במסגרת הגדרת ה Iterator ניתן כמובן להגדיר שדה באובייקט שערכו נשמר:

```
function* idMaker() {
   let index : number = 1;
    yield index++;
    yield index++;
    yield index++;
}
```

מקרה פרטי של קו-ריטוניות הם generators, המחוללים רשימה של ערכים. ובמובן הרחב יותר, הגדרה של מבנה נתונים ע"י קוד, כאשר המידע עצמו מיוצר איבר איבר בזמן אמת רק כאשר הוא נדרש

דוגמא: קו-רוטינה/ג'נטרטור המייצרים את רשימת המספרים מ start ל end:

```
function* range(start, end) {
   for (let i = start; i < end; i++)
        yield i;
}
```

ניתן להשתמש ברשימה הנוצרת באופן הבא:

```
let numList = range(1,4);
while (true) {
   IteratorResult  ir = range.next();
   if (ir.done)
      break;
   console.log(ir.value);
}
```

לא כל כך נוח...
קיים תחביר נוח עבור תבנית זו (syntactic sugar):

```
for (let v of range(1,4))
   console.log(v);
```

בעזרת ג'נרטור ניתן להגדיר רשימה אינסופית:

```
function* naturalNumbers() {
   for (let i = 0; ; i++)
      yield i;
}
```

כדי לעבוד עם רשימות אינסופיות, נגדיר ג'נרטור נוסף המחלץ מרשימה אינסופית את מספר האיברים שאנו רוצים כרגע:

```
function* take(n, gen) {
   for (let x of gen)
      if (n <= 0)
         return;
      n--;
      yield x;
}
```

נדפיס בעזרת מחוללים אלו את שלושת המספרים הטבעיים הראשונים:

```
for (let n of take (3, naturalNumbers()))
    console.log(n); // 0,1,2
```

יתרון הגדרת רשימה ע"י מחולל:
- ניתן להגדיר רשימה אינסופית
- איבר ברשימה מיוצר רק כאשר יש בו שימוש (lazy)
חסרון:
- לא ניתן לגשת מיד, בגישה ישירה, לאיבר באמצע הרשימה. אי אפשר לחזור אחורה. צריך תחילה לייצר את כל האיברים לפניו.

נגדיר את פעולת ה Map כג'נרטור המקבל ג'נרטור ופעולה, ומחזיר בכל שלב את האיבר הבא לאחר ביצוע הפעולה עליו:

```
function* mapGen(generator, f) {
   for (let x of generator) {
      yield f(x);
   }
}

for (let n of take(4, mapGen(naturalNumbers(), x => x * x))) {
   console.log(n); // 0, 1, 4, 9
}
```

באותו אפן, נגדיר את פעולת ה Filter כג'נרטור המקבל ג'נרטור ופרדיקט, ומחזיר בכל שלב את האיבר הבא שמקיים את הפרדיקט:

```
function* filterGen(generator, pred) {
   for (let x of generator) {
      if (pred(x)) {
         yield x;
      }
   }
}

for (let n of take(4, filterGen(naturalNumbers(), x => (x % 2) === 0))) {
   console.log(n); // 0 , 2 , 4 , 6
}
```

כמו תמיד, ניתן להרכיב את שתי הפעולות יחדיו:

```
const evenSquares = filterGen(mapGen(naturalNumbers(), x=> x*x), x=> (x % 2) === 0);
```

```
for (let n of take(4, evenSquares))
   console.log(n);  // 0, 4, 16, 36
```

5.3 Continuation Passing Style

4.3.1 מוטיבציה

נתבונן בפרוצדורה fact המממשת את פעולת העצרת (!) על מספר נתון:

```
(define fact
  (lambda (n)
     (if (= n 1)
        1
        (* n (fact (- n 1))))))
```

הרצה/חישוב של הפעלת הפרוצדורה, כרוכה בשמירת המצב של הפונקציה הקוראת (ה 'AF' שלה) עד שתחזור התוצאה ותמוזג (מכפלה) עם הערך הנוכחי (n).
לדוגמא, עבור (fact 4)

```
(* 4 (* 3 (* 2 1)))
```

עד כדי כך, שקריאה ל (fact 1000) תגרום ל Stack Overflow באינטרפרטר שלנו.

בתרגול ראינו (?) מימוש איטרטיבי לפרוצדורה זו, אשר נמנע מקריאות רקורסיביות שכאלה ('רקורסיית ראש'):

```
(define fact-iter
   (lambda (n acc)
      (if (= n 1)
         acc
         (fact-iter (- n 1) (* n acc)))))

(define fact
   (lambda (n)
      (fact-iter (n 1))))
```

נשים לב, כי בעיצוב זה קיימת קריאה רקורסיבית אך היא הפעולה האחרונה ('רקורסיית זנב'). עבור מקרים אלו,ניתן לממש אופטימיזציה שנמנעת מלשמור את המצב בזמן הקריאה הרקורסיבית.

הקריאות הרקורסיביות עבור (fact 4) יהיו כעת:

```
(fact 4)
(fact-iter 4 1)
(fact-iter 3 4)
(fact-iter 2 12)
(fact-iter 1 24)
24
```

מטרתנו בסעיף זה: להגדיר שיטה כללית הממירה פרוצדורה עם רקורסיית ראש לפרוצדורה שקולה עם רקורסיית זנב. ובמקרה הכללי יותר, ממירה פרוצדורה בה יש קריאה לפרוצדורה אחרת שאינה הפעולה האחרונה, לפרוצדורה שקולה בה כל קריאה לפרוצדורה אחרת הינה הפעולה האחרונה.
השיטה תתבסס על הוספת 'פונקציית המשך' (continuation) כפרמטר לכל פרוצדורה, מעין callback אסינכרוני.

5.3.2 עיצוב פרוצדורה על ידי CPS

ננסה תחילה לאפיין באופן כללי את התרחיש הבעייתי שבו אנו מעוניינים לטפל.

בהינתן ביטוי (AST) נגדיר:

Head Position
תת-ביטוי נמצא ב Head Position אם נדרש לחשבו ולמזגו עם ערך אחר כדי לחשב את ערכו של הביטוי הנתון.

Tail Position
תת-ביטוי נמצא ב Tail Position אם לאחר חישובו לא נדרש פעולה נוספת לשם חישוב הביטוי הנתון.

לדוגמא:

```
(if (> x 2)
   (* x 3)
   x
)
```

בביטוי הנתון יש שלושה תת-ביטויים:

(> x 2) נמצא ב HP, כי לאחר חישובו נדרש חישוב נוסף עבור ערך ביטוי ה if
(* x 3) נמצא ב TP, כי אם נדרש לחשבו זוהי הפעולה האחרונה, לא נדרש לחשב דבר נוסף על מנת למצוא את הערך של ביטוי ה if
x נמצא ב TP, כי אם נדרש לחשבו זוהי הפעולה האחרונה, לא נדרש לחשב דבר נוסף על מנת למצוא את הערך של ביטוי ה if.

ניתן לקבוע מראש, עבור כל סוג של ביטוי בשפה, מה סוג ה position של כל אחד מתתי-ביטוייו:

(define var H) לאחר חישוב הערך, נדרש להוסיף את ה binding לסביבה

(if H T T) לאחר חישוב הבדיקה נדרש לבצע חישוב נוסף של אז/אחרת, שהם הפעולות האחרונות

(lambda (v1 … vn) E E E… E) הערך של ה lambda הוא ה closure העוטף אותה. אין צורך לחשב בשלב זה את הביטויים בגוף הפרוצדורה, כך שהם אינם ב HP אם ב TP

(let ((v1 H) … (vn H)) H H ... H T) ערכי המשתנים הלוקאליים, וכן כל הביטויים הראשונים בגוף הלט אינם הפעולה האחרונה, רק הביטוי האחרון הוא הפעולה האחרונה

(H … H) באפליקציה, לאחר חישוב האופרטור והפרמטרים, נדרש לבצע את ההצבה וחישוב הפרוצדורה

נאמר שביטוי נתון הוא **tail form** אם לא קיימת באף HP קריאה לפרוצדורת משתמש.

פורמלית:

- כל תתי הביטויים (הישירים) ב HP אינם קריאה לפרוצדורת משתמש (שאינה פרימיטיבית)
- כל תתי הביטויים הם בעצמם tail form

דוגמאות:

- `(+ 1 x)` is in tail form.
- `(* (* x x) (+ x x))` is in tail form (combination of primitive applications).
- `(if p x (+ 1 (+ 1 x)))` is in tail form.
- `(f (+ x y))` is in tail form.
- `(+ 1 (f x))` is not in tail form (but `(f x)` is in tail form) because after `(f x)` is computed, the result must be passed to further computation.
- `(if p x (f (- x 1)))` is in tail form.
- `(if (f x) x (f (- x 1)))` is not in tail form - because the head call `(f x)` must be followed by other calls.
- `(lambda (x) (f x))` is in tail form.
- `(lambda (x) (+ 1 (f x)))` is not in tail form because the sub-expression `(+ 1 (f x))` is not in tail form.
- `(lambda (x) (g (f 5)))` is not in tail form.
- `(let ((a 1) (b 2)) (f (+ a b)))` is in tail form
- `(let ((a 1) (b 2)) (f (g a b)))` is not in tail form
- `(let ((a (g 1)) (b 2)) (+ a b))` is not in tail form

המרת פרוצדורה ל tail form

<u>הרעיון הכללי</u>: בכל מקרה בו יש קריאה לפרוצדורת משתמש שאינה הפעולה האחרונה (אינה ה 'return value'), נהפוך אותה להיות הפעולה האחרונה, ע"י ידי הגדרת שאר הקוד כפונקציית ההמשך שלה.

לדוגמא:

```
(+ (f a) (* a a))
→
(f$ a
    (lambda (res) (+ res (* a a))))
```

<u>פרוטוקול ההמרה</u>:

בהינתן פרוצדורה:

- מוסיפים לשם הפרוצדורה המקורית את הסימן $ (קונבנציה)
- מוספים לפרמטרים של הפרוצדורה המקורית פרמטר נוסף - פונקציית ההמשך cont
- מפעילים על כל ערך מוחזר את פונקציית ההמשך

1. כאשר ה-body הוא tail form

- הערך המוחזר אינו הפעלת פרוצדורת משתמש
  נפעיל את פונקציה ההמשך על הערך המוחזר

```
(define square
```

```
  (lambda (x) (* x x)))
```

→

```
(define square$
  (lambda (x cont) (cont (* x x))))


(define add1
   (lambda (x) (+ 1 x)))
```

→

```
(define add1$
   (lambda (x cont) (cont (+ 1 x))))
```

- הערך המוחזר הוא תוצאת הפעלת פרוצדורת משתמש (כלומר זו הפעולה האחרונה) נקרא לגרסת ה CPS שלה עם פונקציית ההמשך

```
(define f1
   (lambda (x y) (square (+ x y)))

(define f1$
   (lambda (x y cont) (square$ (+ x y) cont))

(define f2
   (lambda (x y) (add1 (+ x y)))
```

→

```
(define f2$
   (lambda (x y cont) (add1$ (+ x y) cont))
```

2. כאשר ה-body אינו tail form

```
(define f3
  (lambda (x y) (square (add1 (+ x y))))
```

- מזהים את המקומות הבעייתיים שבהם יש קריאה לפרוצדורה ב HP (אם יש כמה, נבחר אחת מהן (נניח את את הקריאה הפנימית ביותר))
- מסמנים את תוצאת הפעלתו על ידי פרמטר [נניח res]
- משאירים את ההפעלה כפעולה האחרונה ומגדירים את שאר הקוד כפונקציית ההמשך שלה

```
(define f3$
  (lambda (x y cont) (add1$ (+ x y) (lambda (res) (square$ res cont)))))
```

דוגמא נוספת:

```
(define mult
  (lambda (x y) (* x y)))
```

→

```
(define mult$
  (lambda (x y cont) (cont (* x y))))

(define f4
  (lambda (x y) (mult (square x) (add1 y)))
```

→

```
(define f4$
  (lambda (x y cont) (add1$ y (lambda (res1) (mult$ (square x) res1 cont)))))


(define f4$
  (lambda (x y cont) (add1$ y
                              (lambda (res1) (square$ x (lambda (res2) (mult$ res2 res1 cont)))))
```

בחזרה לדוגמת ה fact:

```
(define fact
  (lambda (n)
     (if (= n 1)
        1
        (* n (fact (- n 1))))))
```

→

```
(define fact$
  (lambda (n cont)
     (if (= n 1)
        (cont 1)
        (fact$ (- n 1) (lambda (res) (cont (* n res)))))))
```

נריץ את fact$ עבור החישוב של !2:

- יש להגדיר פונקציית המשך ראשונית (קובעת את ה 'post processing' על התוצאה הסופית של !2)
- באופן טבעי נבחר פונקציית הזהות (lambda(x) x), כי לא נדרש שום עיבוד מאוחר על התוצאה.

```
(fact$ 2 (lambda (x) x))
(fact$ 1 (lambda (res) ((lambda (x) x) (* 2 res)))))))
```

```
((lambda (x) x) (* 2 1))))
```

דוגמא נוספת: sum-odd-squares

```
(define sum-odd-squares
  (lambda (tree)
    (cond
        ((empty? tree) 0)
        ((not (list? tree))
         (if (odd? tree) (square tree) 0))
        (else (+
                (sum-odd-squares (car tree))
                (sum-odd-squares (cdr tree)))))))
```

→

```
(define sum-odd-squares$
  (lambda (tree cont)
    (cond
        ((empty? tree) (cont 0))
        ((not (list? tree))
         (if (odd? tree) (square$ tree cont) (cont 0)))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree)))))))
```

→

```
(define sum-odd-squares$
  (lambda (tree cont)
    (cond ((empty? tree) (cont 0))
        ((not (list? tree))
         (if (odd? tree) (square$ tree cont) (cont 0)))
        (else (sum-odd-squares$ (car tree)
                                (lambda (sum-odd-squares-car-res)
                                  (+ sum-odd-squares-car-res (sum-odd-squares (cdr tree))) ))))))
```

→

```
(define sum-odd-squares$
  (lambda (tree cont)
    (cond ((empty? tree) (cont 0))
        ((not (list? tree))
         (if (odd? tree) (square$ tree cont) (cont 0)))
        (else (sum-odd-squares$ (car tree)
                                (lambda (sum-odd-squares-car-res)
                                  (sum-odd-squares$ (cdr tree)
                                    (lambda (sum-odd-square-cdr-res)
```

```
(cont (+ sum-odd-square-car-res sum-odd-square-cdr-res)))))))))
```

גרסת CPS לפונקציות מסדר גבוה: map, filter

```
(define map
  (lambda (f list)
    (if (empty? list)
        list
        (cons (f (car list))
              (map f (cdr list))))))
```

→

```
(define map$
  (lambda (f$ list cont)
    (if (empty? list)
        (cont list)
        (f$ (car list)
            (lambda (f-res)
              (cons f-res (map f (cdr list))) )))))
```

→

```
(define map$
  (lambda (f$ list cont)
    (if (empty? list)
        (cont list)
        (f$ (car list)
           (lambda (f-car-res)
            (map$ f$ (cdr list)
              (lambda (map-f-cdr-res)
                (cont (cons f-car-res map-f-cdr-res))))))))
```

```
(define filter
  (lambda (pred? list)
    (cond
        ((empty? list) list)
        ((pred? (car list))
         (cons (car list) (filter pred? (cdr list))))
        (else (filter pred? (cdr list))))))
```

→

```
(define filter$
  (lambda (pred?$ list cont)
    (cond ((empty? list) (cont list))
```

```
      (else (pred?$ (car list)
            (lambda (pred-res)
              (cond (pred-res (cons (car list) (filter pred? (cdr list))))
                  (else (filter pred? (cdr list)))) ))))))
```

→

```
(define filter$
  (lambda (pred?$ list cont)
    (cond ((empty? list) (cont list))
      (else (pred?$ (car list)
            (lambda (pred-res)
              (cond (pred-res (filter$ pred?$ (cdr list)
                                        (lambda (filter-cdr-res)
                                          (cont (cons (car list) filter-cdr-res)))))
                  (else (filter$ pred?$ (cdr list) cont))))))))
```

<u>הגדרת יחס שקילות בין פרוצדורה לגרסת ה CPS שלה</u>:

ראינו כבר הגדרת יחס שקילות עבור שתי פונקציות.
יש להגדיר מחדש יחס זה, עבור פונקציה וגרסת ה CPS שלה (שהרי לגרסת ה CPS יש פרמטר נוסף של פונקציית ההמשך).

בהינתן פונקציה (f x1…xn), וגרסת ה CPS שלה (f$ x1…xn cont)
נאמר כי הפונקציות שקולות, אם לכל סדרת אופרנדים a1,…,an ופונקציית המשך המשך k מתקיים:

(f$ a1…an k) = (k (f a1…an))

fact, fact$ שקולות (אפשר להוכיח באינדוקציה על n)
בפרט:

(fact$ 3 square) = (square (fact 3))

מימוש אינטרפרטר המבצע כל קריאה לפרוצדורה כפעולה אחרונה

מאחר ומדובר בהמרה תחבירית, ניתן היה לממש פרוצדורה (ב TS) המקבלת AST של פרוצדורה (ProcExp), ומחזירה AST של גרסת ה CPS שלה.

אנו נבחר בדרך אחרת: נממש את התנהגות ה CPS בקוד של האינטרפרטר עצמו.
כלומר, נעצב את האינטרפרטר כך שהוא אף פעם לא קורא לפרוצדורה כשהיא לא הפעולה האחרונה.

קוד: L6-eval.ts, הפרוצדורה evalCont

מימוש אופטימיזציית קריאות הזנב

האינטרפרטר/הקומפיילר של שפות תכנות שונות כולל אופטימיזציה המבטיחה שאם יש קריאה לפרוצדורה שהיא הפעולה האחרונה, אזי לא נשמר המצב של הפונקציה הקוראת.
למרבה המבוכה, אופטימיזציה זו אינה ממומשת ב JavaScript...

כלומר, בכל מקרה יישמר המצב של הפונקציה הקוראת.
מבחינה מעשית, חישוב הקריאה ((fact$ 1000 (lambda (x) x) יגרום ל Stack Overflow באינטרפרטר.

נקודת אור אחת: כאשר הפעולה האחרונה היא קריאה לפרוצדורה <u>ללא פרמטרים</u>, האופטימיזציה ממומשת אפילו ב JavaScript.

🡸 נעדכן את האינטרפטר כך שכל הקריאות שלו לפרוצדורות יהיו ללא פרמטרים, ע"י תבנית העיצוב fetch/decode/execute:

- נגדיר משתנים 'גלובליים' עבור המידע הקיים בדרך כלל ב AF: הפרמטרים של הפרוצדורה המופעלת כעת, הערך המוחזר שלה, כתובת החזרה (לאיזו שורה בקוד צריך לחזור בתום הפרוצדורה)
- כל קריאה לפרוצדורה תיפתח בהשמת הפרמטרים למשתנים אלו, וכן לכתובת החזרה (ה pc הנוכחי)
- בתום כל פרוצדורה נעתיק את הערך החוזר מהמשתנה שלו.

קוד: L7c-eval.ts
Success-Fail Continuations

ניתן להרחיב את הקונספט של פונקציות ההמשך, לתבנית עיצוב בה פרוצדורה מקבלת שתי פונקציות המשך: האחת להפעלה במקרה של הצלחה (עיבוד מאוחר של תוצאת הפרוצדורה, בדומה להנדלר ה then בתבנית ה Promise), והשניה להפעלה במקרה של כישלון (בדומה לפונקציית ה catch בתבנית ה Promise)

דוגמאות:

1. סכימת מספרים ברשימה הטרוגנית

‘((1 3) 2)
← 6
‘((a 3) 2)
← error

```
;; Signature: sumlist(li)
;; Purpose: Sum the elements of a number list.
;; If the list includes a non-number element -- produce an error.
;; Type: [List -> Number union ???]
(define sumlist
  (lambda (li)
    (cond ((empty? li) 0)
          ((number? (car li)) (+ (car li) (sumlist (cdr li))))
          (else (error "non numeric value!"))
  )
)
```

גרסת CPS, עם פונקציות המשך להצלחה ולכישלון:

```
(define sumlist$
   (lambda (li succ-cont fail-cont)
```

```
      (cond ((empty? li) (succ-cont 0))
                ((number? (car li))
                  (sumlist$ (cdr li)
                    (lambda (sum-cdr)
                      (succ-cont (+ (car li) sum-cdr)))
                    fail-cont))
               (else (fail-cont)))
   )
)

(define sumlist2
  (lambda (li)
      (sumlist$ li
               (lambda (x) x)
               (lambda () (display "non numeric value!")))
   )
)
```

2. חיפוש המספר הזוגי השמאלי ביותר בעץ

```
;; Signature: leftmost-even(tree)
;; Purpose: Find the leftmost even leaf of an unlabeled tree whose leaves are labeled by
numbers.
;; If no leaf is even, return #f.
;; Type: [List<Number> -> Number union Boolean]
;; Examples: (leftmost-even '((1 2) (3 4 5))) ==> 2
;;                 (leftmost-even '((1 1) (3 3) 5)) ==> #f
(define leftmost-even
  (lambda (tree)
    (cond ((empty? tree) #f)
               ((not (list? tree)) ;; leaf?
                 (if (even? tree) tree #f))
               (else ;;; Composite tree
                 (let ((res-first (leftmost-even (car tree))))
                   (if res-first
                       res-first
                     (leftmost-even (cdr tree))))))
  )
)
```

גרסת CPS עם פונקציות המשך להצלחה ולכישלון

```
(define leftmost-even$
  (lambda (tree succ-cont fail-cont)
    (cond ((empty-tree? tree) (fail-cont))
         ((not (list? tree))
          (if (even? tree)
             (succ-cont tree)
```

```
            (fail-cont)))
        (else ; Composite tree
          (leftmost-even$ (car tree)
                succ-cont
                (lambda () (leftmost-even$ (cdr tree)
                        succ-cont
                    fail-cont)))))))

(define leftmost-even2
  (lambda (tree)
    (leftmost-even$ tree
          (lambda (x) x)
          (lambda () #f))))
```

5.4 רשימות עצלות וקו-רוטינות ב-L

כזכור, ראינו כי ניתן להגדיר ב JS מחוללים (generators) המייצרים בפרט רשימות אינסופיות (כמו המחולל naturalNumbers המייצר את רשימת המספרים השלמים, מספר אחרי מספר ע"פ דרישה). כמו כן כי המחולל הוא מקרה פרטי של קונספט הקו-רוטינה – ביצוע פרוצדורה שלב אחרי שלב (מתודת ה next של ממשק ה Iterator)

ניישם שני דברים אלו בשפה L (החל מ 2L)

4.4.1 רשימות עצלות (Lazy Lists)

רשימה עצלה מייצגת רשימה כזוג של האיבר הבא, וקוד המשך לייצור שאר איברי הרשימה.

Lzl(T) = Empty-Lzl | Pair(T, (Empty → Lzl(T))

ADT:

```
(define epmty-lzl? empty?)
(define cons-lzl cons)
(define head car)
(define tail
   (lambda (lzl)
      ((cdr lzl))
   )
)
```

דוגמאות:

1. רשימת המספרים השלמים (החל ממספר נתון n)

```
;; Signature: integers-from(n)
;; Type: [number -> LZL(number)]
```

```
(define integers-from
  (lambda (n)
     (cons-lzl n (lambda () (integers-from (+ n 1)))

(define numbers (integers-from 0))
(head numbers)
→ 0
(head (tail numbers))
→ 1
(head (tail (tail numbers)))
→ 2
```

2. ייצור n האברים הראשונים ברשימה עצלה נתונה

```
;; Signature: take(lz-lst,n)
;; Type: [LzL*Number -> List]
;; If n > length(lz-lst) then the result is lz-lst as a List
(define take
  (lambda (lz-lst n)
    (if (or (= n 0) (empty-lzl? lz-lst))
      empty-lzl
      (cons (head lz-lst)
            (take (tail lz-lst) (- n 1))))))

(take 2 numbers)
→ '(0 1)
```

3. החזרת המספר ה-nי ברשימה עצלה נתונה

```
; Signature: nth(lz-lst,n)
;; Type: [LzL<T>*Number -> T]
;; Pre-condition: n < length(lz-lst)
(define nth
  (lambda (lz-lst n)
    (if (= n 0)
       (head lz-lst)
       (nth (tail lz-lst) (- n 1)))))

(nth numbers 2)
→ 2
```

4. רשימה עצלה של 1ים

```
(define ones (cons-lzl 1 (lambda () ones)))
```

5. רשימה עצלה של !n

```
(define facts-gen
  (lambda ()
```

```
   (letrec ((loop (lambda (n fact-n)
                 (cons-lzl fact-n
                        (lambda () (loop (+ n 1)
                                    (* (+ n 1) fact-n)))))))
     (loop 1 1))))

> (take (facts-gen) 6)
--> '(1 2 6 24 120 720)
```

6. בניית רשימות עצלות מרשימות עצלות קיימות

```
;; Signature: lz-lst-add(lz1,lz2)
;; Type: [LzL(Number) * LzL(Number) -> LzL(number)]
(define lz-lst-add
  (lambda (lz1 lz2)
   (cond ((empty-lzl? lz1) lz2)
         ((empty-lzl? lz2) lz1)
         (else (cons-lzl (+ (head lz1) (head lz2))
                         (lambda () (lzl-add (tail lz1) (tail lz2))))))))

(define integers
  (cons-lzl 0
            (lambda () (lzl-add ones integers))))

0,
   11111111…
       +
   012 3….


(define fib-numbers
  (cons-lzl 0
         (lambda () (cons-lzl 1
                        (lambda ()
                          (lzl-add (tail fib-numbers) fib-numbers))))))

0,
   1,
     1,2,3…
       +
     0,1,2…


(take fib-numbers 7)
--> ’(0 1 1 2 3 5 8)
```

7. פעולות על רשימות עצלות

```
;; Signature: lz-lst-append(lz1, lz2)
;; Type: [Lzl(T) * Lzl(T) -> Lzl(T)]
(define lzl-append
  (lambda (lz1 lz2)
    (if (empty-lzl? lz1)
        lz2
        (cons-lzl (head lz1)
                  (lambda () (lzl-append (tail lz1) lz2))))))
```

במימוש זה, אם הרשימה הראשונה היא אינסופית, לא נגיע לעולם לאיבר הראשון ברשימה השניה:

```
> (take (lzl-append (integers-from 100) fibs) 7)
'(100 101 102 103 104 105 106)
```

הפרוצדורה interleave משרשרת לסירוגין אברים משתי הרשימות:

```
;; Signature: interleave(lz1, lz2)
;; Type: [Lzl(T) *Lzl(T) -> Lzl(T)]
(define interleave
  (lambda (lz1 lz2)
    (if (empty-lzl? lz1)
        lz2
        (cons-lzl (head lz1)
                  (lambda () (interleave lz2 (tail lz1)))))))
```

```
> (take (interleave (integers-from 100) fibs) 8)
'(100 0 101 1 102 1 103 2)
```

8. פונקציות מסדר גבוה על רשימות עצלות

```
;; Signature: lzl-map(f, lz)
;; Type: [[T1 -> T2] * Lzl(T1) -> Lzl(T2)]
(define lzl-map
  (lambda (f lzl)
    (if (empty-lzl? lzl)
        lzl
        (cons-lzl (f (head lzl))
                  (lambda () (lzl-map f (tail lzl)))))))
```

```
;; Signature: lz-lst-filter(p,lz)
;; Type: [[T -> Boolean] * Lzl(T) -> LzL(T)]
(define lzl-filter
  (lambda (p lzl)
    (cond ((empty-lzl? lzl) lzl)
          ((p (head lzl)) (cons-lzl (head lzl)
                                    (lambda () (lzl-filter p (tail lzl)))))
          (else (lzl-filter p (tail lzl))))))
```

9. רשימה עצלה של המספרים הראשוניים

```
(define prime?
  (lambda (n)
    (letrec ((iter (lambda (lz)
                     (cond ((> (sqr (head lz)) n) #t)
                           ((divisible? n (head lz)) #f)
                           (else (iter (tail lz)))))))
      (iter primes))))

(define primes
  (cons-lzl 2 (lambda () (lzl-filter prime? (integers-from 3)))))

(take primes 6)
--> '(2 3 5 7 11 13)
```

[גרסה יעילה יותר הנמנעת מחישובים מיותרים – חומר קריאה

The second definition we present avoids the redundancy of the computation above. It implements the sieve algorithm. The lazy-list of primes can be created as follows:

Start with the integers lazy-list: [2,3,4,5,....].
Select the first prime: 2.
Filter the current lazy-list from all multiples of 2: [2,3,5,7,9,...]
Select the next element on the list: 3.
Filter the current lazy-list from all multiples of 3: [2,3,5,6,11,13,17,...].
i-th step: Select the next element on the list: k. Surely it is a prime, since it is not a multiplication of any smaller integer.
Filter the current lazy-list from all multiples of k.
All elements of the resulting lazy-list are primes, and all primes are in the resulting lazy-list.

```
;; Signature: sieve(lzl)
;; Type: [Lzl(Number) -> Lzl(Number)]
(define sieve
  (lambda (lzl)
    (cons-lzl (head lzl)
          (lambda ()
            (sieve (lzl-filter (lambda (x) (not (divisible? x (head lzl))))
                       (tail lzl)))))))

(define primes1 (sieve (integers-from 2)))
(take primes1 7)
--> '(2 3 5 7 11 13 17)
```

]

5.4.2 מימוש קו-רוטינות (co-routines) בעזרת רשימות עצלות

1. מבנה נתונים

ניתן להשתמש בקונספט של רשימות עצלות, כדי להגדיר קו-רוטינות, כלומר פרוצדורה המתבצעת שלב אחרי שלב.

הרעיון הכללי: קו-רוטינה תהיה רשימה עצלה, שהאבר הראשון שלה הוא ביטוי שערכו הוא הערך המוחזר מהשלב הראשון, וקוד ההמשך יבצע את שאר השלבים (כלומר, הוא יהיה פונקציה המחזירה זוג של ביטוי שערכו הוא הערך המוחזר מהשלב השני ופונקציית המשך). פונקציית ההמשך של השלב האחרון תחזיר 'done'.

בנאי: הפרוצדורה yield - מקבלת את תוצאת השלב הנוכחי ואת פונקציית ההמשך המקודדת את השלבים הבאים, ומחזירה אותם כ 'Iterator' - <u>זוג של תוצאת השלב הנוכחי ופונקציית המשך המקודדת את השלבים הבאים</u>.

```
(define yield
   (lambda (step cont-steps)
      (cons-lzl step cont-steps)))
```

'מתודות':

הערך של השלב הנוכחי/האחרון

```
(define iter->value
  (lambda (iter)
   (if (iter->done? iter)
      iter
      (car iter))))

(define iter->done?
  (lambda (iter)
   (eq? iter 'done)))

(define iter->next
  (lambda (iter)
   (if (iter->done? iter)
      iter
      (let ((cont (cdr iter)))
       (if (eq? cont 'done)
          cont
          (cont))))))
```

2. הגדרת generator

```
(define g
   (yield 1
        (lambda ()
         (yield 2
```

```
            (lambda ()
              (yield 3
                     'done)))))))
```

3. שימוש ב generator

```
(iter->value g)
→1
(iter->value (iter->next g))
→2
(iter->value (iter->next (iter->next g)))
→3
(iter->value (iter->next (iter->next (iter->next g))))
→ ‘done
```

הפרוצדורה iter->take מחזירה את התוצאה של n השלבים הראשונים של איטרטור נתון:

```
;; Purpose: return the first n elements generated by an iterator as a list.
;; Type: [Iterator(T) * number -> List(T)]
;; Returns a list of up to n elements - can be less if the generator is done before.
;; On a done iterator, returns an empty list.
(define iter->take
  (lambda (iter n)
    (letrec ((loop (lambda (iter n res-lst)
                 (if (<= n 0)
                    res-lst
                    (if (iter->done? iter)
                       res-lst
                       (loop (iter->next iter)
                             (- n 1)
                             (concat res-lst (iter->value iter))))))))
     (loop iter n ‘())))))

(iter->take g 2)
→ ‘(1 2)
```

6. תכנות לוגי

עד כה עסקנו בעיקר בדגם אחד של שפות תכנות – שפות פונקציונאליות (ואף מימשנו חמש שפות L1-L5).
בפרק זה, נתוודע לדגם אחר של שפות תכנות (אותו הזכרנו כבר בשיעור המבוא) – שפות לוגיות.
במסגרת זו נגדיר שתי שפות, האחת (שפה לוגית רציונלית) שאינה Turing-complete, והשניה (שפה לוגית) שכן.
בדומה לשפות ה -L שהיו תת-שפה של שפה אמיתית (scheme) אך כתבנו להן אינטרפרטר משלנו, גם השפות הלוגיות שנגדיר הן תת-שפה של שפה אמיתית (Prolog) וגם במקרה זה נכתוב אינטרפרטר בעצמנו, משלנו (נכתוב אותו ב-L5).
כהרגלנו, נתמקד <u>בתחביר ובסמנטיקה</u> של השפה הלוגית.

5.1 שפה לוגית רציונלית

5.1.1 תחביר

השפה הלוגית הרציונלית מורכב מביטויים ונוסחאות (יחסים בין ביטויים).
הנוסחאות יכולות להיות אטומיות - עובדות (facts), או מורכבות – חוקים (rules).

ביטוי יכול להיות סמל (מתחיל באות קטנה), או משתנה (מתחיל באות גדולה או ב '_')

ניתן לחשוב על הסמלים כמייצגים ישויות בעולם: abraham, computer
המשתנים, כמו כל משתנה בשפת תכנות, מייצגים ערך כלשהו שעשוי להתקבל בהמשך, כלומר סמל מסוים:
Person, _Device

עובדה מתארת יחס בוליאני בין ביטויים. יחס שערכו הוא true.
לדוגמא:

```
parent(abraham, isaac).
male(abraham).
female(sara).
female(hagar).
parent(abraham, ishmael).
parent(sara,isaac).
parent(hagar, ishmael).
```

תוכנית לוגית מורכבת מביטויים ונוסחאות, ומשאילתא עליהם:

```
?- parent(hagar, ishmael)
true
```

האופן שבו מתבצע ההסק, או המעבר משאילתא לערך, מוגדר על ידי הסמנטיקה של השפה. נעסוק בכך בהמשך (קשור למושג היוניפיקציה שפגשנו כבר בפרק על הטיפוסים).

```
?- parent(abraham, X)
X = isaac
X = ishmael
```

במקרה זה, הערך המוחזר הוא הצבות המשתנים שעבורן השאילתא תהיה בעלת ערך אמת.

```
?-parent(ishmael, X)
false
```

אם נוסיף לתכנית את העובדה

```
parent(ishmael, nevayot).
```

אז נקבל עבור אותה שאילתא:

```
?-parent(ishmael, X)
X  = nevayot
```

מי הם ההורים של ישמעאל?

```
?- parent(X, ishmael), parent(Y, ishmael)
X = hagar, Y = hagar
X = abraham, Y = abraham
X = abraham, Y = hagar
X = hagar, Y = abraham
```

אם רוצים ש X ו Y יהיו שונים:

```
?- parent(X, ishmael), parent(Y, ishmael), X \= Y
X = abraham, Y = hagar
X = hagar, Y = abraham
```

אם רוצים 'אבא' ו'אמא':

```
?- parent(X, ishmael), male(X) , parent(Y, ishmael), female(Y)
X = abraham, Y = hagar
```

האם יש בן של שרה שהוא הורה של מישהו אחר?

```
?- parent(sara, X), parent(X,Y)
false
```

נוסיף את הבנים של יצחק לאוסף העובדות:

```
parent(isaac,jacob).
parent(isaac,esav).

?- parent(sara, X), parent(X,Y)
X = isaac, Y = jacob
X = isaac, Y = esav
```

<u>חוקים</u> קובעים יחס מורכב יותר בין ביטויים. ניתן לחשוב עליהם כמו 'כללי הסק'.

לדוגמא: חוק המגדיר את יחס ה'אבהות' ויחס ה'אימהות' (שאינם מוגדרים במפורש על ידי העובדות, אך ניתן להסיקן בהתאם ל'כוונת המחוקק')

```
father(Dad, Child):- parent(Dad,Child), male(Dad).
mother(Mom, Child):- parent(Mom,Child), female(Mom).
```

משמעות החוק (המכונה גם 'פרוצדורה') היא: לכל המשתנים האפשריים בחוק (Dad/Mom, Child בדוגמא שלנו), אם מתקיים ה RHS (המכונה body) אז ה LHS (המכונה Head) נכון / ערכו אמת.

<u>הערה</u>: נשים לב לכך, כי עובדה היא למעשה חוק שה RHS שלו הוא true.

```
parent(hagar, ishmael).
```
➔
```
parent(hagar,ishmael):-true.
```

נוסיף כמה עובדות:

```
parent(rivka, jacob).
parent(rivka, esav).
female(rivka).
```

יש למצוא אמא של שני ילדים:

```
?-mother(M,C1), mother(M,C2), C1 \= C2
M = rivka, C1 = jacob, C2 = esav
M = rivka, C1 = esav, C2 = jacob
```

נגדיר חוק נוסף (ואחרון בשלב זה) המגדיר את יחס הורה קדמון ויוצאי חלציו:

```
ancestor(A, D):- parent(A,D).
ancestor(A, D):- parent(A, Person), ancestor(Person, D).
```

מיהם צאצאיו של אברהם:

```
?-ancestor(abraham, D)
D =  isaac
D = ishmael
D = jacob [Person = isaac]
D = esav [ Person = isaac]
D = nevayot [ Person = ishmael]
```

מיהם הוריו הקדמונים של עשו?

```
?-ancestor(A, esav)
A =  isaac
A = rivka
A = abraham [Person = isaac]
A = sara [Person = isaac]
```

שאלה: מה היה קורה אם היינו הופכים את סדר החוקים, כלומר ממקמים את החוק של מקרה הקצה אחרי החוק השני? נראה בהמשך, כי אנו עשויים להיקלע ללואה אינסופית.

5.1.2 סמנטיקה (אופרציונאלית)

יש להגדיר כיצד הופכת השאילתא לערך (לתשובה / הצבה)

5.1.2.1 תשתית: יוניפיקציה

1. הגדרות (תזכורת)

<u>הצבה</u> (Substitution)

אוסף של זוגות, כאשר האיבר הראשון בכל זוג הוא <u>משתנה לוגי</u>, והשני <u>ביטוי לוגי</u> (תחת האילוץ שהמשתנה משמאל לא מופיע בביטוי מימין)

לדוגמא:

{ X = abraham, Z = isaac }
{X = abraham, **Y = Y** } **Error**

<u>הפעלת הצבה</u> (substitution application)

בפעולה זו אנו מפרשים נוסחה לוגית נתונה לאור הצבה קיימת: החלפת משתנים בנוסחה המוגדרים בהצבה, על ידי הצבת ה RHS שלהם בנוסחה.

parent(X,Y) { X = abraham, Y = isaac } = parent(abraham, isaac)

<u>הרכבת הצבות</u> (substitution combination)

נתונות שתי הצבות S1,S2

יש להרכיב אותן להצבה אחת: S1° S2

כזכור, העיקרון הכללי: לוקחים כבסיס את ההצבה הראשונה (1S), ומפרשים אותה ע"פ ההצבה השניה (2S), כל עוד זה לא סותר.

- הפעלת 2S על 1S
- הוספת הזוגות ב 2S שה LHS שלהן אינו מוגדר ב 1S
- הסרת חוקי זהות X=X

{ X = Y, Z = V } ° { Y = abraham, D = issac, Z = X} = { X = abraham, Z = V, Y = abraham, D = isaac}

<u>ספציפיות</u>

נאמר כי נוסחה אטומית (כלומר לא חוק שלם / כלל הסק עם head ו body) A' ספציפית יותר מנוסחה אטומית A, אם יש הצבה S כך ש: A ° S = A'
כלומר, A' היא פירוש מסוים של A על פי תרחיש/הצבה מסוים/מת.

לדוגמא:

A = parent(X,Y)
A' = parent(abraham, Y)

A' ספציפי יותר מ A כי הוא פרשנות מסוימת שלו, ופורמאלית: A ° { X = abraham } = A'

<u>מאחד</u> (unifier)

המאחד S של שתי נוסחאות אטומיות A,A' הוא הצבה ההופכת אותם לשווים: A ° S = A' ° S

לדוגמא:

A = parent(X,Z)
A' = parent(abraham, Y)

S1 = { X = abraham, Y = isaac, Z = isaac}
S2 ={ X = abraham, Y = Z }

S1 ו S2 הם מאחדים של A ו 'A, כאשר S2 כללי יותר.
המאחד הכללי ביותר מכונה כזכור MGU – במקרה שלנו S2.

2. אלגוריתם למציאת ה MGU – יוניפיקציה

נתונים שני ביטויים, יש למצוא את המאחד הכללי ביותר, כלומר את התנאים הבסיסיים ביותר (=ההצבה הכללית ביותר) ההופכים אותם לשווים.
נגדיר משוואה של שני הביטויים, ונפעיל את אלגוריתם פתרון המשוואות מהפרק על הטיפוסים.

דוגמא: יש לאחד את הביטויים

p(X,a,X,W)
p(Y,Y,Z,Z)

ניצור משוואה: p(X,a,X,W) = p(Y,Y,Z,Z)

ונריץ את אלגוריתם פתרון המשוואות:

<u>אתחול</u>

אוסף המשוואות: [p(X,a,X,W) = p(Y,Y,Z,Z)]
הצבה: {}

בחירת משוואה מהאוסף: p(X,a,X,W) = p(Y,Y,Z,Z)
'פרשנות' המשוואה על ידי הפעלת ההצבה הנוכחית על שתי אגפיה: p(X,a,X,W) = p(Y,Y,Z,Z)

'מקרה ג' – שני האגפים הם ביטויים מורכבים בעלי אותו מבנה (אותו פרדיקט, אותו מספר פרמטרים)
← פירוק למשוואת קטנות יותר

אוסף המשוואות: [X=Y, Y=a, X=Z, W=Z]
הצבה: {}

בחירת משוואה מהאוסף: X = Y
'פרשנות' המשוואה על ידי הפעלת ההצבה הנוכחית על שתי אגפיה: X = Y

'מקרה ב' – אחד הצדדים הוא משתנה לוגי
← הוספת המשוואה להצבה: {X=Y} = {X=Y} ° {}

אוסף המשוואות: [Y=a, X=Z, W=Z]
הצבה: { X=Y }

בחירת משוואה מהאוסף: Y = a
'פרשנות' המשוואה על ידי הפעלת ההצבה הנוכחית על שתי אגפיה: Y=a

'מקרה ב' – אחד הצדדים הוא משתנה לוגי

← הוספת המשוואה להצבה: { X=Y } ° {Y=a} = {X=a, Y=a}

אוסף המשוואות: [X=Z, W=Z]
הצבה: {X=a, Y=a}

בחירת משוואה מהאוסף: X = Z
'פרשנות' המשוואה על ידי הפעלת ההצבה הנוכחית על שתי אגפיה: a=Z
'מקרה ב' – אחד הצדדים הוא משתנה לוגי

← הוספת המשוואה להצבה: { X=a, Y=a } ° {Z=a} = {X=a, Y=a, Z = a}

אוסף המשוואות: [W=Z]
הצבה: {X=a, Y=a, Z=a}

בחירת משוואה מהאוסף: W = Z
'פרשנות' המשוואה על ידי הפעלת ההצבה הנוכחית על שתי אגפיה: W=a
'מקרה ב' – אחד הצדדים הוא משתנה לוגי

← הוספת המשוואה להצבה: { X=a, Y=a, Z=a } ° {W=a} = {X=a, Y=a, Z = a, W=a}

אוסף המשוואות: []
הצבה: {X=a, Y=a, Z=a, W=a}

← המאחד הכללי ביותר (mgu) הוא ההצבה {X=a, Y=a, Z=a, W=a}

5.1.2.2 תיאור אלגוריתם חישוב השאילתא

נתונה תוכנית הכוללת אוסף נוסחאות (עובדות, כללי הסק) ושאילתא.
יש למצוא את הצבות המשתנים בשאילתא ההופכות אותה לערך אמת.

<u>לדוגמא</u>

נגדיר חוק חדש:

```
son(X,Y) :- parent(Y,X), male(X).
```

נוסיף את העובדות הבאות:

```
parent(jacob, josef).
parent(jacob, dan).
parent(jacob, dina).

male(josef).
male(dan).
```

female(dina).

יש לחשב את השאילתה:

?- son(S, Jacob)

אנו מצפים שהאלגוריתם יתן שתי תשובות/הצבות אפשריות:

{ S = josef }
{ S = dan }

<u>הרעיון הכללי</u>:

- נעבור על כל חלק של השאילתא (במקרה שלנו יש רק אחד (son(S, jacob), כל חלק שכזה מכונה goal, ועבור כל חלק נמצא את התנאים/ההצבות שהופכות אותו לערך true. ואחר כך נרכיב יחדיו את ההצבות של כל חלק, כדי לקבל את התנאים לערך אמת עבור השאילתא כולה.
- מציאת ההצבה ההופכת כל חלק/goal לערך אמת:
  - נעבור על כל החוקים הרלבנטיים עבור ה goal הנתון, כלומר כל החוקים שקיימת הצבה המאחדת את ה LHS שלהם עם ה goal, ונחליף את ה goal ב RHS שלהם.
- במהלך מציאת התשובה, לא נמצא רק את ההצבה הנדרשת, אלא גם נקודד את הדרכים השונות לפיתרון, ע"י בניית מבנה נתונים המכונה 'עץ הוכחה'.

<u>תשתית לאלגוריתם</u>

1. יש לקבוע את הסדר בו עוברים על חלקי השאילתא השונים (על ה goals)
הפונקציה Gsel מקבלת שאילתא, כלומר סדרת goals, ומחזירה את ה goal הבא לפיתוח.
לדוגמא:

?- parent(X,jacob), female(X)

Gsel([parent(X,jacob), female(X)]) = female(X)

2. יש לקבוע את הסדר בו בוחרים את החוקים מהתוכנית

הפונקציה Rsel מקבלת goal ותוכנית, ומחזירה את אוסף החוקים הרלבטיים ל goal באופן ממוין, כאשר כל חוק ניתן יחד עם ההצבה המתאימה (כלומר יד עם התנאים שבהם הוא מתאים ל goal ע"פ היוניפיקציה)

Rsel(son(S, jacob), Program) = [<son(X,Y):-parent(Y,X),male(X) , { S=X, Y=jacob} >]

3. מבנה הנתונים של עץ ההוכחה

כזכור, האלגוריתם לא רק מוצא את ההצבות ההופכות את השאילתא לערך אמת, אלא גם מתאר את הדרך על ידי 'עץ הוכחה'.

<u>קודקודים</u>: כל קודקוד מציין שאילתא (וכן סימון מה ה goal הבא בשאילתא זו לפיתוח). בתחילת האלגוריתם יהיה קודקוד אחד, השורש, המייצג את שאילתת המשתמש/ת, בהמשך תפורק שאילתא זו לשאילתות פשוטות יותר תוך יצירת קודקודים נוספים המייצגים אותן.

<u>צלעות</u>: כל צלע מכוונת מייצגת חוק לפיתוח השאילתא (ממנה הצלע יוצאת) לשאילתא חדשה ופשוטה יותר (הקודקוד אליה הצלע מגיעה). הצלע תכלול גם את ההצבה שעל פיה נבחר החוק.

פעולות:

make_node(label)

התווית כוללת את השאילתא

add_branch(node, edge_lable, branch)

הוספת תת העץ branch לעץ הנתון node, עם צלע שהתווית שלה edge_label כוללת את החוק הנבחר ואת ההצבה שבשמה הוא נבחר.

label(node)

החזרת התווית של קודקוד נתון, כלומר את השאילתא שהוא מייצג

האלגוריתם

קלט:

Query: Q = ?-G_1,…,G_n
Program: P [a set of (numbered) rules]
Gsel [selects the next goal for a given query]
Rsel [selects the relevant rules for a given goal and program]

פלט:

Proof Tree

תאור האלגוריתם:

proof_tree(make_node(Q))

כאשר

```
proof_tree(node)
   query := label(node)
   if query is '?- true,…,true'
        mark node as 'success'
   else
        goal := Gsel(query)
         [rename variables]
         rules := Rsel(goal,P)
         if empty(rules)
              mark node as 'failure'
         else
              for <rule,sub> in rules
                   new_query := replace(query, goal, body(rule)) ° sub
                   add_branch(node, <rule,sub>, proof_tree(make_node(new_query)))
   return node
```

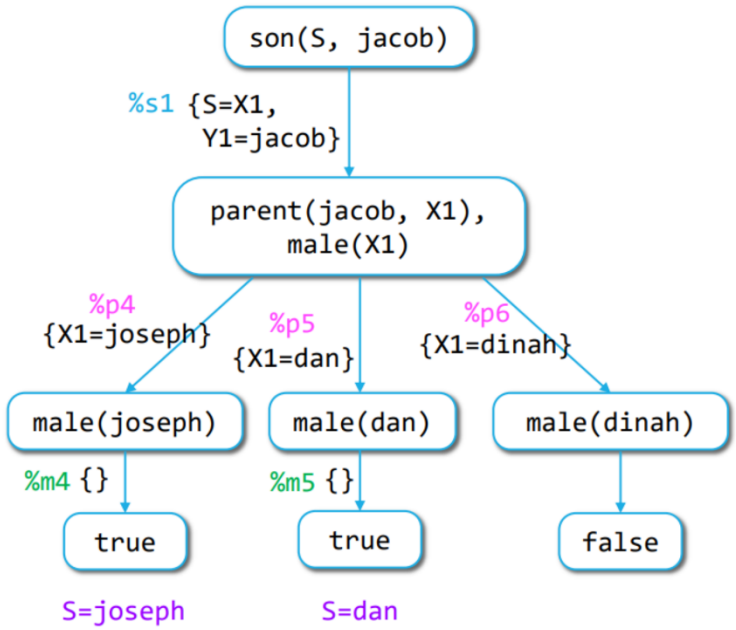
בסוף התהליך, נקבל עץ שחלק מהעלים שלו הם 'קודקודי הצלחה', וחלק הם 'קודקודי כישלון'. כל מסלול מהשורש לקודקוד הצלחה מייצג אפשרות אחת לפיתרון. כאשר הפיתרון הוא הרכבת כל ההצבות הניתנות בצלעות של המסלול הנתון.

```
parent(abraham, isaac).  %p1
parent(isaac, jacob).    %p2
parent(sarah, isaac).    %p3
parent(jacob, joseph).   %p4
parent(jacob, dan).      %p5
parent(jacob, dinah).    %p6

male(abraham).           %m1
male(isaac).             %m2
male(jacob).             %m3
male(joseph).            %m4
male(dan).               %m5

son(X,Y) :- parent(Y,X),  %s1
            male(X).
```

```
?- son(S,jacob).
```



5.1.2.3 מושגים ומאפיינים

<u>מושגים</u>

- <u>מסלול הצלחה</u> (successful computation path) הינו מסלול בעץ ההוכחה, מהשורש ועד לקודקוד הצלחה.
- <u>מסלול כישלון</u> (failure computation path) הינו מסלול בעץ ההוכחה, מהשורש ועד לקודקוד כישלון.
- עץ הכולל מסלול הצלחה (אחד לכל הפחות) נקרא <u>עץ הצלחה</u> (success tree).
- עץ שכל מסלוליו הם מסלולי כישלון נקרא <u>עץ כישלון</u> (failure tree).
- עץ הכולל מסלול אינסופי מכונה <u>עץ אינסופי</u> (בדרך כלל בשל רקורסיה).
- עץ שכל מסלוליו סופיים מכונה <u>עץ סופי</u>.
- <u>עץ הצלחה סופי</u> הוא עץ סופי עם מסלול הצלחה.
- <u>עץ כישלון סופי</u> הוא עץ סופי שכל מסלוליו הם מסלולי כישלון.
- <u>עץ הצלחה אינסופי</u> הוא עץ אינסופי שיש בו מסלול הצלחה (סופי).

<u>מאפיינים</u>

שאילתא Q הינה <u>ברת-הוכחה</u> (provable) מתוכנית נתונה P, מסומן על ידי P |-- Q, אם עבור Gsel ו Rsel כל שהם (=לא משנה סדר המעבר על השאילתא והחוקים) נוצר עץ הצלחה.

לשאילתא ולתכנית נתונות יש עץ הוכחה יחיד, ללא תלות בסדר החישוב (Gsel, Rsel), עד כדי איזומורפיות.

תוכנית בשפה הלוגית-רציונאלית היא כריעה (decidable). כלומר, ניתן מראש לענות על השאלה, האם שאילתא נתונה היא ברת-הוכחה ביחס לתוכנית נתונה.
[לכאורה זה סותר את האבחנה, כי עשוי להתפתח עץ הוכחה אינסופי, כך שלא ניתן לדעת האם נגיע אי פעם לקודקוד הצלחה. אך אופי השפה (מספר חוקים סופי, מספר מטרות סופי בשאילתא) גוזר, שאם חזרנו לקודקוד שיצרנו כבר (עם אותה שאילתא), ניתן להסיק שזה מסלול אינסופי, עם לולאה, שלעולם לא יגיע לקודקוד הצלחה]

- השפה הלוגית-רציונאלית אינה Turing-Complete.

5.2 שפה לוגית (לא רציונאלית)

נרחיב את השפה הלוגית-רציונאלית כך שתכלול אפשרות להגדרת ביטוי מורכב (המציין 'אובייקט'), נכנה ביטוי/אובייקט שכזה functor.

5.2.1 תחביר

עד כה, היו שני סוגים של ביטויים: סמלים (abraham) ומשתנים (X), וכן הביטויים/ערכים הפרימיטיביים true, false.
הדרך היחידה להרכיב אותם היתה ע"י הגדרת יחס: parent(abraham, ishmael)

בשפה הלוגית החדשה יש literal expression המאפשר להגדיר ביטוי שהוא מבנה, 'אובייקט'.
לדוגמא:

time(june,mon)
location(beersheva, israel)

ניתן לחשוב על שני פקטורים אלו כמו על שני אובייקטים בג'אווה, האחד מופע של המחלקה Time והשני מופע של המחלקה Location.

נשים לב לכך, שהמבנה של הפנקטור זהה למבנה של נוסחה אטומית. האבחנה שלהם ניתנת על ההקשר שבהם הם מופיעים: פנקטורים ממומקים תמיד 'בתוך סוגרים', כ'פרמטר' של משהו אחר.

לדוגמא:

א.

time(june,mon)
location(beersheva,israel)

← נוסחאות: היחס time מתקיים (ערך אמת) עבור june ו mon. היחס location מתקיים (ערך אמת) עבור beersheva ו israel.

ב.

course(ppl, time(june,mon), location(beersheva,israel))

← פנקורים: היחס course מתקיים עבור הישויות ppl, time(june,mon), location(beersheva,israel)

5.2.2 סמנטיקה

ללא שינוי, כמו קודם, אותו אלגוריתם למציאת ההצבות ההופכות שאילתא נתונה לערך אמת (בהינתן תכנית).

נדרש רק לטפל שאפשרות של פנקטור בשני מקרים:

- כאשר מרכיבים הצבות, כזכור, נדרש לוודא שהמשתנה המופיע בצד שמאל בביטוי הלוגי אינו מופיע גם מצד ימין. עד כה, האפשרות היחידה לכך היתה ביטוי מהצורה X = X, כעת יש גם את האפשרות (שצריך לפסול) של פנקטורים, לדוגמא: X = location(X,israel)

- בפתרון המשוואות (במסגרת היוניפיקציה), ב'מקרה ג', בו יש שני ביטויים מורכבים בעלי אותו מבנה, עד כה האפשרות היחידה היתה שני יחסים בעלי אותו מבנה: parent(X,isaac) = parent(abraham,Y) ← X = abraham, Y = isaac. כעת יש מקרה נוסף של משוואה בין ביטויים מורכבים, עבור פנקטורים: location(beersheva,Y) = location(X,israel) ← X = beersheva, Y = israel.

השפה הלוגית החדשה היא Turing-Complete, אך היא אינה כריעה [לא ניתן להניח שאם הגענו שוב לאותו קודקוד אין טעם להמשיך, כי למרות שמספר החוקים והמטרות סופי, מספר הפנקטורים אינו סופי. כפי שנראה להלן].

5.2.3 ייצוג מבני נתונים ומספרים בשפה הלוגית: עצים, רשימות, מספרים שלמים

1. עצים

נגדיר עץ בינארי ע"פ החוק/יחס הבא:

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left), binary_tree(Right).
```

נגדיר את יחס ה'חברות' בעץ בניארי:

```
tree_memeber(X,tree(X,_,_)).
tree_member(X, tree(_,Left,_)) :- tree_memeber(X,Left).
tree_member(X, tree(_,_,Right)) :- tree_memeber(X,Right).
```

שאילתות:

```
?- tree_member(g(X),
                tree(g(a),
                     tree(g(b), void,void),
                     tree(f(a), void,void)))

X = a
X = b

?- tree_member(a, Tree)
```

נקבל אינסוף פתרונות (הצבות למשתנה Tree) – כל העצים בעולם שיש להם קודקוד כלשהו עם הערך a

```
Tree = tree(a,void,void)
Tree = (_X1,tree(a,void,void), _X2)
…
```

2. רשימות

ניתן, בעזרת פנקטורים, להגדיר רשימות.
כזכור, כדי להגדיר רשימה באופן אינדוקטיבי, צריך:

- להגדיר את הרשימה הריקה
- להגדיר זוג איברים, כאשר האבר השני הוא בעצמו רשימה

ב-L3 הגדרנו לשם כך שני פרימיטיביים: ()‘, cons
בשפה הלוגית נעשה זאת בעזרת הסמל empty והפנקטור cons
כעת ניתן להגדיר באופן אינדוקטיבי את היחס 'רשימה':

```
list(empty).
list(cons(X,Xs)) :- list(Xs).
```

בשפה L3 הגדרנו תחביר נוח לרשימות, כמו (3 2 1)‘, נעשה זאת אף לשפה הלוגית שלנו:

הרשימה הריקה: [] ← empty
רשימה עם איברים: [c,b,a] ← cons(c, cons(b, cons(a,empty))))

לשם נוחות נוסיף אופרטור פרימיטיבי '|', המחלק רשימה לשני חלקים: [X|Xs] – X מייצג את האיבר הראשון ברשימה ו Xs את כל השאר. באותו אופן: [X,Y|L] – Y,X הם שני האיברים הראשונים ברשימה ו L היא שאר הרשימה.

נגדיר מספר יחסים:

חברות ברשימה

```
member(X, [X|_Xs]).
member(X,[_Y|Ys]) :- member(X, Ys).
```

```
?- member(a, [b, c, a, d]).
true

?- member(X, [a,b,c]).
X = a
X = b
X = c

?- member(a, List).
List = [a]
List = [_X1,a]
…
any list containing ‘a’
```

שרשור רשימות

```
append([], Xs, Xs):-list(Xs).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs), list(Ys).

?- append([a,b], [c], L).
L = [a,b,c]
```

```
?- append(Xs, [c,d], [a,b,c,d]).
Xs = [a,b]

?- append(Xs, Ys, [a,b,c]).
Xs = [a,b,c]   Ys = []
Xs = [a,b]   Ys = [c]
Xs = [a]   Ys = [b,c]
Xs = []   Ys = [a,b,c]
```

הגדרת יחסים שונים ברשימות על בסיס יחס השרשור append

```
% (a) List prefix and suffix:
prefix(Xs, Ys) :- append(Xs, Zs, Ys).
suffix(Xs, Ys) :- append(Zs, Xs, Ys).

% (b) Redefine member:
member(X, Ys) :- append(Zs, [X | Xs], Ys).

% (c) Adjacent list elements:
adjacent(X, Y, Zs) :- append(Ws, [X, Y | Ys], Zs).

% (d) Last element of a list:
last(X, Ys) :- append(Xs, [X], Ys).
```

הגדרת היחס בין רשימה ורשימה הפוכה (עם אותם איברים בסדר הפוך)

```
reverse([], []).
reverse([H | T], R) :- reverse(T, S), append(S, [H], R).

?- reverse([a, b, c], R).
R=[ c, b, a]
```

גרסה איטרטיבית, ללא רקורסיות ראש, תוך שימוש באקומולטור:

```
reverse(Xs, Ys):- reverse_help(Xs,[],Ys).

reverse_help([X | Xs], Acc, Ys) :- reverse_help(Xs, [X | Acc], Ys).
reverse_help([], Ys, Ys).

?- reverse([a,b,c], R).
reverse_help([a,b,c],[],R)
reverse_help([b,c],[a|[]],R)
reverse_help([c],[b|[a|[]]],R)
reverse_help([],[c|[b|[a|[]]]],R)
R = [c|[b|[a|[]]]] = [c,b,a]
```

3. מספרים

בשפה הלוגית יש רק סמלים, אין מושג אריתמטי של מספר (0 לדוגמא הוא הסמל '0', לא הערך החשבוני 0)
נגדיר את המספרים הטבעיים באופן לוגי בעזרת פנקטור (Church Numeral Encoding):
- נייצג את המספר 0 על ידי סמל פרימיטיבי חדש 0
- נייצג את המספר 1 על ידי הפנקטור s(0) [שמשמעו עבורנו הוא המספר הבא אחרי 0]
- נייצג את המספר 2 על ידי הפנקטור s(s(0)) [שמשמעו עבורנו הוא המספר הבא אחרי המספר הבא אחרי 0]
- וכן הלאה

כעת ניתן להגדיר את יחס 'מספר טבעי':

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).


?- natural_number(s(s(0))).
true
```

נגדיר את פעולות החיבור והכפל בין מספרים שלמים בייצוג זה כיחסים, וכן את יחס ההשוואה (קטן מ):

```
% Signature: plus(X, Y, Z)/3
% Purpose: Z is the sum of X and Y.
plus(X, 0, X) :- natural_number(X).
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).    %    x + (y+1) = (z+1) ← x + y = z

?- plus(s(0), 0, s(0)).
true.

?- plus(X, s(0), s(s(0)).
X=s(0)

?- plus(X, Y, s(s(0))).
X=0, Y=s(s(0))
X=s(0), Y=s(0)
X=s(s(0)), Y=0

% Signature: times(X,Y,Z)/3
% Purpose: Z = X*Y
times(0, X, 0) :- natural_number(X).
times(s(X), Y, Z) :- times(X, Y, XY), plus(XY, Y, Z).

% Signature: le(X,Y)/2
% Purpose: X is less or equal Y.
le(0, X) :- natural_number(X).
le(s(X), s(Z)) :- le(X, Z).
```

5.3 מימוש האינטרפרטר (ב L4)

1. ייצוג תחבירי

LP-ast.rkt

- תחביר קונקרטי ומופשט
- מימוש התחביר ב-L4
  - ייצוג כל אובייקט תחבירי כרשימה (מה שהיה ב interface JS)
    - [ה-tag מופיע במקום הראשון ברשימה, ולאחריו השדות]
  - פרוצדורת הבנאי make המייצרת רשימה בהתאם
  - 'getters' לחילוץ השדות על פי מיקומם הרשימה (קונבנציית ה '<-' בשם הפרוצדורה)
  - [פרדיקט טיפוס (על בסיס התג בראש הרשימה)]
- אין כרגע פארסר (תש כוחנו...), אך ניתן להגדיר תוכנית לוגית בעזרת הבנאים (ראו קבצי ה-tests)

2. מימוש ההצבה

substitution-ADT.rkt

פרוצדורות: apply, combine

מימוש ב-L4 של L5-substitution-adt.ts מהפרק על הטיפוסים.

3. יוניפיקציה

unify.rkt

פרוצדורה: unify_formulas

מימוש ב-L4 של אלגוריתם פתרון מהשוואות מהפרק על הטיפוסים.

4. עצים עצלים

lazy-tree-ADT.rkt

- ייצוג עץ עצל כזוג של שורש ופונקציה ליצירת בנים.
- בנאי: expand-lzt
- ה'מתודה' lzt->branches מייצרת את רשימת הבנים של הקודקוד הנתון, על ידי הפעלת פונקציית ייצור הבנים של קודקוד זה.
- filter
  מקבלת עץ עצל ופרדיקט סינון, ומחזירה את הקודקודים שמספקים את הפרדיקט.
  שתי גרסאות:
  lzt-filter מחזירה את הקודקודים כרשימה רגילה
  lzt-filter-lzl מחזירה את הקודקודים כרשימה עצלה

5. מימוש אלגוריתם פתרון השאילתא

answer-query.rkt

- הגדרת פונקציית ייצור הבנים LP-node-expander, על פי האלגוריתם (=על פי רשימת החוקים-סביבות החוזרים מ Rsel עבור המטרה הנבחרת על פי Gsel, כאשר כל חוק-הצבה מייצר שאילתא פשוטה יותר)
- הגדרת שורש עץ ההוכחה עם השאילתא המקורית (במימוש זה, שומרים לשם נוחות בקודקוד לא רק את השאילתא אלא גם את הרכבת ההצבות על הצלעות מהשורש ועד הקודקוד)
- 'בניית עץ ההוכחה' (בגרסה העצלה לא מבצעים את פיתוח העץ כעת אלא מסתפקים רק בשורש הכולל את פונקציית ייצור הבנים)
- חילוץ התשובות ע"י ביצוע filter עם פרדיקט המסנן קודקודים שאינם קודקודי הצלחה (בגרסת answer-query מופעלת גרסת lzt-filter המחזירה את כל הקודקודים כרשימה, בגרסת answer-query-lzl מופעלת גרסת lzt-filter-lzl המחזירה את הקודקודים כרשימה עצלה, כלומר פתרון אחר פתרון)
- כל קודקוד הצלחה שחוזר מ filter מכיל כבר את הרכבת ההצבות מהשורש אליו. נותר רק לחלץ מההצבה רק את המשתנים שהופיעו בשאילתא המקורית.