עקרונות שפות תכנות, 2020

~ppl202

מני אדלר

שעות קבלה: שני 16-18

1. מבוא, תחביר סמנטיקה וטיפוסים ב TypeScript
   1. מבוא
      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 / שפה שנמציא.
* עיצוב מטה-תוכניות (Meta-Programming-)

תוכניות המקבלות כקלט תוכניות אחרות, ועושות איתן משהו:

* + אינטרפרטר: מקבל תוכנית, מריץ/מחשב אותה.
  + הסק טיפוסים: מקבלת תוכנית ללא טיפוסים, מחזירה את התוכנית מהקלט עם טיפוסים.
  + המרת קוד: קבלת תוכנית בשפה אחת, והחזרת תוכנית בשפה אחרת.
  + 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. דגמי שפות תכנות

* שפות אימפרטיביות (Imperative Languages)

תוכנית = רצף של פקודות

דוגמאות: Java, C++

* שפות הצהרתיות (Declarative Languages)  
  תוכנית = הצהרה על הדבר המבוקש  
  דוגמא: SQL
* שפות מבניות (Structural Languages)  
  תחביר השפה כולל מבנים מקוננים (כך שאין צורך ב goto): if-else, do-while  
  דוגמאות: Java, C++
* שפות פרוצדוראליות (Procedural Languages)  
  תחביר השפה מאפשר להגדיר קוד כפרצודרה, כך שניתן לקרוא ממקומות שונים בקוד.

דוגמאות: Java, C++

* שפות פונקציונאליות (Functional Languages)
  + התוכנית היא ביטוי ולא רצף של פקודות

הרצת התוכנית היא חישוב של הביטוי 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);

* + 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 כפרטר 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, וכללי הסק

הרצת התוכנית הינה ביצוע של שאילתות על העובדות וההסקים

* + 1. 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
* התאמה לערכים אחרים דורשת שינוי של הקוד
* תחביר השפה לא מלמד על כך שמדוובר בדפוס חוזר

🡸 שפה עם מבנה:

* משתנים, פרמטרים
* 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 דורש שינוי של הקוד (המשתנה numbers)
* 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);

}

חסרונות:

* הפרוצדורות print ו squares מבצעות עדיין שני דברים – מעבר על מערך ופעולה על כל איבר.
* יש מצב משותף שמתעדכן (מערך המספרים numbers): מקשה על מקביליות, אימות קוד, אופטימיזציות...

🡸 תכנות פונקציונאלי

R.map(x=>console.log(x), R.map(x=>x\*x,[0,1,2,3]));

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 האאיברים שצלחו את הפונקציה הבוליאנית (כלומר, שהפונקציה הבוליאנית החזירה עבורם ערך אמת).

console.log(R.filter(x => x % 2 == 0, [0,1,2,3]));

R.map(x=>console.log(x), R.filter(x => x % 2 == 0,[0,1,2,3]));

דוגמא נוספת (בתרגול): הפונקציה reduce

ניתן אף להרכיב שתי פונקציות לפונקציה מורכבת חדשה, בעזרת הפונקציה מסדר גבוה compose.

function square(x) {

return x\*x;

}

function isEven(x) {

return x % 2 == 0;

}

let isEvenSquareF = R.compose(R.filter(isEven), R.map(square));

console.log(isEvenSquareF([0,1,2,3]));

שקילות של פונקציות

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 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?

כן – ראו הוכחה קצרה [כאן](https://www.cs.bgu.ac.il/~ppl202/wiki.files/ppl_chapter1.pdf), עמוד 24 למטה עד עמוד 25.

* 1. תחביר, סמנטיקה, וטיפוסים ב TypeScript
     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

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-פרומינג המקבל כקלט תוכנית בשפה, וכן עבור מי שרוצה להוכיח שקילות בין תוכניות וכו'.

סמנטיקה פורמאלית, מגדירה את האופן שבו יש לפרש/לחשב ביטוי ביטוי 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
  + החזרת הערך ע"פ סוג הביטוי

let x = 12;

(x > 7) ? (x \* 3) : 9;

1. ביטויים וערכים

כזכור, המונח ביטוי מתייחס למבנים השונים בשפת התכנות, והערכים הם המשמעו 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

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. טיפוסים
       1. טיפוס כקבוצה

נגדיר טיפוס כקבוצה של ערכים

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 {1-10, 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 הגרת טיפוס חדש ע"י פעולת disjoint union

ניתן אף לאפיין קשרים בין טיפוסים, כיחסים בין קבוצות:

ספיציפיות (טיפוס ותת-טיפוס): 1T הוא תת-טיפוס של 2T אם הקבוצה 1T מוכלת בקבוצה 2T. לדוגמא: חיה – כלב

זרות: הטיפוס 1T זר לטיפוס 2T, אם החיתוך של הקבוצות 1T ו 2T ריק. לדוגמא: יונקים – ציפורים.

* + - 1. Type Checking

מנגנון 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.

x = ‘a’

If (x > 7)…

כדאי להגדיר שטיפוסים:

* בדיקת תאימות טיפוסים מראש (לא בזמן ריצה)
* הקוד קריא יותר
* תורם לעיצוב נכון יותר
  + - 1. 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
    - 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):

console.log(1);  
[number]

console.log(‘1’);  
[string]

console.log([1,2,3]);  
[object]

console.log(typeof({ 'a' : 1, 'b' : true}));

[object]

ניתן לברר באופן מדוייק 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]

הגדרת ערכים בשפת התכנות

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

{a:1, b:true}

ב Java לא קייימת 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 movieLists = [

{

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 }]

}

]

}

];

הגדרת טיפוסים ב TS

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.

מתן שם לטיפוס

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

type <type name> = <type>

לדוגמא:

type StringArray = 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[]}

טיפוס לפונקציות:

const add : (x : number, y : number) => number =   
 (x : number, y : number) : number => x+y;

add(1,2)

3

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

שימוש בפעולות על קבוצות לשם הגדרת טיפוסים חדשים:

* 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

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]](#footnote-1), name : string, address : string }

interface Variable { tag : “variable”, name : string, address : string}

let p : Person = { name : 'yossi', address : 'beer sheva'},

v : 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";

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 => {

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} = {(0,0), (1,0), (2,0), (2,1), (3,1)}

מבנים רקורסיביים

ראינו כי בעזרת ה interface ניתן להגדיר טיפוסים רקורסיביים (Person, Student, Course)

נבחן דוגמא נוספת:

interface BinTree {

root : number,

left : BinTree | null,

right : BinTree | null

}

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 | null,

right : BinTree | null

}

let tree : BinTree = {

root : 2,

left : { root : 1, left : null, right : null },

right : { root : 3, left : null, right : null }

}

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}

}

טיפוסים גנריים

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 | 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 => x\*x;

const squareTree : (t: BinTree<number>) => BinTree<number> = (t) =>

(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)};

squareTree(bt2);

}

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](https://www.cs.bgu.ac.il/~ppl202/wiki.files/class/notebook/1.4TypesAndOperations.html).

יחסי טיפוסים בין טיפוסים גנריים:

BinTree<number> ------ instantiation ---------- BinTree<T>

BinTree<string> -------- disjoint ------ BinTree<number>

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

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

Clusure

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 12.

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 'סגור' 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.

תאימות טיפוסים

* נאמר כי טיפוס T2 *תואם (compatible)* לטיפוס T1, אם ניתן להשים ערך מטיפוס T2 למשתנה מטיפוס T1. (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];

x = arr // 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 תואם למערך עם טיפוס תואם

let pArr : Person[] = …,  
 sArr : Student[] = …;

pArr = sArr // correct  
 sArr = pArr // incorrect

* + מילון תואם למילון אם יש לו לפחות אותו מספר שדות

let person : { name : string } = …,  
 student : { name : string, univ :string } = …;

person = student // correct  
 student = person // incorrect

* + פונקציה תואמת לפונקציה כאשר:
    - מספר הפרמטרים זהה
    - הערך המוחזר תואם
    - הפרמטרים תואמים באופן הפוך

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](https://medium.com/@michalskoczylas/covariance-contravariance-and-a-little-bit-of-typescript-2e61f41f6f68). 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.

1. תחביר וסמנטיקה
   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]

1. אופני הרכבה

כיצד לבנות ביטוי מורכב מביטויים פשוטים .

1. אופני הפשטה

כיצד ניתן לתאר ביטויים מורכבים ביחידה עצמאית / באופן פשוט יותר.

* 1. הגדרת שפות תכנות בסיסיות
     1. השפה L1

**I תחביר השפה 1L**

1. ביטויים פרימיטיביים

אטומיים:

Literal numbers: 1, 2, 3,…

Literal boolean: #t, #f

Primitive procedures: +, -, \*, /, >, <, =

1. אופני הרכבה

בשפה 1L 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) (-3 2) 2)  
🡪 54

ג. אופני הפשטה

ב 1L ניתן לתת שם לביטוי, בעזרת ה*אופרטור המיוחד* define

(define pi 3.14)  
…  
(\* pi 2)  
🡪 6.28

הקשר בין שם לבין ביטוי/ערך מכונה binding.

II סמנטיקה

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 (דיון להלן)
* משתנים

ע"פ הערך שהוגדר עבורו ב define (כלומר lookup)

* אופרטור מיוחד

לא מחושבים (משמשים רק לזיהוי מבנה מיוחד)

1. חישוב ביטויים מורכבים לערך

* צורות מיוחדות (special form, סוגרים שבמקום השמאלי נמצא אופרטור מיוחד, כמו define) מחושבות כל אחת על פי משמעות הספציפית.

בשפה 1L 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
  + חישוב כל אחד מהפרמטרים
  + הפעלת הפרוצדורה/אופרטור על הפרמטרים

(+ (\* 2 3) 7)

🡪13

(= 2 (+ 1 1))

🡪 #t

נקודות למחשבה

* 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

חסרונות השפה 1L:

* 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 מסויים (כל המשתנים גלובליים)
  + 1. השפה 2L

בשפה 2L 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) (\* x x))  
 3  
)

🡪

9

(square 3)  
🡪  
9

1. נגדיר את הצורה המיוחדת if

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

(if (> 3 1) #t #f)

(if (= (square 3) 9) (+ 1 2) (- 3 4))

II סמנטיקה

1. אופן החישוב של lambda

* הגדרת הפרוצדורה היא יצירת closure הכולל בתוכו את: רשימת הפרמטרים, גוף הפרוצדורה, ואת הסביבה הנוכחית (נדון בהמשך).
* הפעלת פרוצדורה:
  + החלפת כל מופע של פרמטר בגוף הפרוצדורה בערך שנשלח עבורו
  + חישוב הביטויים בגוף הפרוצדורה
  + הערך החוזר הוא ערכו של הביטוי האחרון

1. 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!**

תכנות ב 2L: 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 (נוסיף כזה בתרגיל 2)
* לא ניתן לשנות ערך של משתנה (לא בהכרח חיסרון, שפה פונקציונאלית)
* מבנה נתונים מורכב (כמו רשימה) כחלק מהשפה
* משתנים לוקאליים

🡸 בשפה 3L יתווספו מבנה נתונים בסיסי ומשתנים לוקאליים.

* + 1. השפה 3L

בשפה זו נוסיף ביטויים עבור ערכים מורכביים, ומשתנים מקומיים.

1. ערכים מורכבים

ב JS היו שני סוגים של ערכים מורכבים: מערכים ומילונים.

ב 3L נגדיר מבנה בסיסי יותר: זוג ביטויים/ערכים

תחביר:

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), תאור 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)

בעזרת זוגות, ניתן לתאר בנוחות יחסית כל מבנה נתונים.

לשם נוחות יתר, נרחיב את השפה כך שתתמוך ברשימות.

הגדרה אינדוקטיבית של רשימה:

* רשימה יכולה להיות הרשימה הריקה
* או זוג של ביטוי ורשימה

הרשימה 1,2,3 לדוגמא תהיה:

(cons 1 (cons 2 (cons 3 ‘())))

🡸 נוסיף לשפה פרימיטיב חדש עבור הרשימה הריקה:

אופרטורים: empty-list, empty?

Literal-expression: ‘()

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

(define list?

(lambda (l)

(or (empty? l)

(and (cons? l)

(list? (cdr l))))))

(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).

נרחבי 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 (l)

(if (empty? l)

0

(+ 1 (length (cdr l))))))

(define nth

(lambda (l)

(if (empty? l)

'()

(if (= n 0)

(car l)

(nth (- n 1) (cdr l))))))

1. 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)**))))

* 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)**))))

* חיסרון: חישוב חוזר של הקריאה לפרצודורה (כל פעם שהביטוי מופיע)
* פתרון: הגדת הביטוי כמשתנה לוקאלי.

🡸 הצורה המיוחדת let

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
* חישוב ה bodyע"פ הסביבה הנוכחית, הכוללת אתת המשתנים הלוקאליים

אבחנה: 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)

)

)

)

לסיום, לשם נוחות, נרחיב במקצת את אוסף סוגי הערכים בשפה 3L:

String “a b c”

Symbol ‘green

2.2.3.3 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 l. Return the list of results

(define map

(lambda (f l)

(if (empty? l)

'()

(cons (f (car l))

(map f (cdr l))))))

דוגמת ריצה:

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

l = '(2 3)

f = (lambda (x) (\* x x))

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

l = '(3)

f = (lambda (X) (\* x x))

(cons 9 (map f '()))

l = '()

f = (lambda (X) (\* x x))

'()

--> (cons 4 (cons 9 '()))

ב. filter

;;Signature: filter(pred,l)

;;Type: [(T1->Boolean)\*List(T1) -> List(T1) ]

;;Purpose: Return the elemnts of l which satisfy the predicate pred

(define filter

(lambda (pred l)

(if (empty? L)

‘()

(if (pred (car l))

(cons (car l) (filter pred (cdr l)))

(filter pred (cdr l))))))

(filter (lambda (x) (> x 0)) ‘(1 -1))

-> ‘(1)

1. reduce

;;Signature: reduce(reducer,init,l)

;;Type: [(T1\*T2->T2)\*T2\*List(T1)-> T2]

;;Purpose: Combine the values of l, starting with init, according to the reducer

(define reduce

(lambda (reducer init l)

(if (empty? l)

init

(reducer (car l)

(reduce reducer init (cdr l))))))

(reduce + 0 '(1 2))

reducer: +

init: 0

l: '(1 2)

(+ 1 (reduce + 0 '(2)))

reducer: +

init: 0

l: '(2)

(+ 2 (reduce + 0 '()))

reducer: +

init: 0

l: '()

0

(+ 1 (+ 2 0))

;;Signature: reduce(reducer,init,l)

;;Type: [(T1\*T2->T2)\*T2\*List(T1)-> T2]

;;Purpose: Combine the values of l, starting with init, according to the reducer

(define reduce2

(lambda (reducer init l)

(if (empty? l)

init

(reduce2 reducer

(reducer (car l) init)

(cdr l)))))

(reduce2 + 0 '(1 2))

reducer: +

init: 0

l: '(1 2)

(reduce2 + 1 '(2))

reducer: +

init: 1

l: '(2)

(reduce2 + 3 '())

3

האם reduce, reduce2 שקולות?

ראינו כי סדר הפעלת פונקציית המיזוג (reducer) על איברי הרשימה שונה בשני המימושים.

🡨 אם היא אינה אסוציטיבית הפונקציות אינן שקולות.

1. 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+8) + 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 גבוהה, המקבלת את next ו f (כיצד להתקדם, ומה מבצעים על כל איבר) כפרמטרים:

;; Signature: sum(term, 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 next a b)

(if (> a b)

0

(+ (f a)

(sum f next (next a) b)))))

(define sum-integers

(lambda (a b)

(sum (lambda (x) x) (lambda (x) (+ x 1)) a b)))

(define sum-cubes

(lambda (a b)

(sum cube (lambda (x) (+ x 1)) a b)))

(define pi-sum

(lambda (a b)

(sum (lambda (x) (/ 1 (\* x (+ x 2)))) (lambda (n) (+ n 4)) a 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+dex/2]dx

כאשר n את התחום [a,b] ליחידות 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 (lambda (x) (+ x dx) (+ a (/ dx 2)) b)

dx)))

דוגמא אחרונה: חישוב קירוב לנגזרת

הפונקציה 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:

(let [(f1 (derive square 0.001))

(f2 (derive square 0.1)) ]

(f1 2) --> 4.000999999999699 (Real value is 4)

(f2 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,n)

;; Type: [ (Number->Number) \* Number -> (Number->Number) ]

;; Purpose: construct a function that computes a numerical approximation of the nth derivative of f

(define nth-deriv

(lambda (f n)

(lambda (x)

(if (= n 0)

(f x)

( (nth-deriv (derive f 0.0001) (- n 1)) x)))))

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

(define nth-deriv-early

(lambda (f n)

(if (= n 0)

f

(derive (nth-deriv-early f (- n 1)) 0.0001))))

* 1. הגדרה פורמאלית של תחביר, ומימושו
     1. מבוא

התחביר:

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

דוגמאות:

* שפה טבעית (עברית, אנגלית): המילון, אוצר המילים בשפה, כולל הטיות

כסא

שולחן

הכסא

שולחני

שולחנות

...

* פרוטוקול תקשורת בין תהליכים

לדוגמא, פרוטוקול STOMP מתרגיל 3 ב SPL

login, send, ‘abc’, \n, \0….

* השפה 2L

(, ), define, lambda, if, 3, #f,…

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

* שפה טבעית

S

N V N

דני הלך הביתה

הלך

מושא

נושא

דני הביתה

* פרוקטוקל תקשורת

Login danny\n

Send cars ‘…’\n

* L2

(if (> x 3) 4 5)

🡸 הגדרת תחביר באופן פורמאלי, מבוססת 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

1. 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 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’]

🡪

if-exp

alt

test

then

app-exp num-exp num-exp

rands

rator

prim-op [var-ref num-exp] 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.

* + 1. 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

1. 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 בפשה, 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 האינטרפרטר)

<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.

1. Parser

מימוש הפארסר ממוקד בקובץ [L3-ast.ts](https://github.com/bguppl/interpreters/blob/master/src/L3/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](https://github.com/bguppl/interpreters/blob/master/src/shared/result.ts) 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’]]

]

program

exps

]def-exp def-exp if-exp proc-exp let-exp[

body

body

val

args

val

bindings

alt

then

test

var

var

var-decl num-exp var-decl lit-exp app-exp num-exp num-exp [var-decl] [app-exp] […] […]

rands

rands

rator

rator

var

val

val

val

val

var

var

x 5 s abc prim-op [var-ref num-exp] 3 4 y prim-op [var-ref var-ref]

op

op

val

var

var

var

> x 2 + x y

* + 1. 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) ? 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;

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

הגדרות:

משתנה x מופיע חופשי (occurs free) בביטוי E אם ורק אם:

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

משתנה x מופיע כבול (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 :

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) :

isPrimOp(e) ? false :

isAppExp(e) ? occursFree(v, e.rator) ||

some((rand) => occursFree(v, rand), e.rands) :

isDefineExp(e) ? (v !== e.var.var) && occursFree(v, e.val) :

isLetExp(e) ? false : // TODO

isProgram(e) ? false : // TODO

false;

1. מציאת כל המשתנים שיש אליהם ref בביטוי נתון
2. 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

)

1. הוספת 'כתובת לקסיקאלית' לכל התייחסות למשתנה בתוכנית (מעין צביע 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 x)) 1)

(lambda (x y)

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

([+ free] [x : 0 0] [x : 0 0])) 1)

מימוש:

* הרחבת מבני הנתונים בדקדוק עם מבנים 'ספציפיים' יותר ל VarRef
  + FreeVar
  + LexicalAddress
  + CExpLA (הרחבה של Cexp כך שמתאפשר גם VarRef מסוג FreeVar ו LexicalAddress)
* פונקציות עזר

crossContour

מבוצעת כאשר נכנסים לרמה חדשה של הגדרת משתנים:

* 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

בהינתן מילון 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]

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)) :

isProcExpLA(exp) ? visitProc(exp, addresses) :

isAppExpLA(exp) ?

safe2((rator: CExpLA, rands: CExpLA[]) => makeOk(makeAppExpLA(rator, rands)))

(visit(exp.rator, addresses), mapResult(rand => visit(rand, addresses), exp.rands)) :

exp;

return visit(exp, []);

};

לחמש הפעולות שהדגמנו יש תבנית משותפת:

* פעולות על עץ תחבירי מופשט
  + פעולה בהתאם לטיפוס
    - מקרה קצה
      * חישוב גובה: כל קודקוד מגדיל את הגובה באחד
      * האם משתנה חופשי בביטוי: VarRef - כן
      * מציאת התייחסות למשתנים: - VarRefהרשימה [v]
      * הוספת כתובת לקסיקאלית:
        + VarDecl (lambda, define, let) – הרחבת מילון ההגדרות
        + VarRef – החלפת ה VarRef ב LexicalAddrees/FreeVar בהתאם למילון
      * המרת let: החלפת ה LetExp ב AppExp
    - קריאה רקורסיבית לתתי ביטויים
      * איחוד תוצאות
        + חישוב גובה: + max
        + האם משתנה חופשי בביטוי: or
        + מציאת התייחסות למשתנים: איחוד קבוצות
        + הוספת כתובת לקסיקאלית: בניית קודקוד תואם חדש עם תתי הביטויים החדשים
        + המרת let: בניית קודקוד תואם חדש עם תתי הביטויים החדשים
  1. הגדרה פורמלית של סמנטיקה ומימושה באינטרפרטר

עד כה, הגדרנו באופן פורמאלי את התחביר עבור השפות L1-L3, ומיממשנו 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
  + 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.

גישה ב: 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.

2L

נוספו שני סוגי ביטויים: IfExp, ProcExp

Value = Number | Boolean | PrimOp | undefined | Closure

L3

נוסף המבנה LetExp, וכן הגדרת זוגות ורשימות, סמלים ומחרוזות.

יש להגדיר ערך המייצג זוגות ורשימות.

באופן מעשי, ערך זה מייצג כל סוג של היררכיית ערכים.

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 = Number | Boolean | PrimOp | undefined | Closure | SExp

1. הגדרה פורמאלית של חוקי החישוב

נגדיר באופן פורמאלי, עבור כל סוג של ביטוי, את האופן שבו הוא הופך לערך:

Atomic Expressions

Numbers ;; 7

eval(<num-exp> exp, env) 🡺 exp.val

Booleans ;; #t

eval(<bool-exp> exp, env) 🡺 exp.val

Primitive operators ;; +

eval(<prim-op> exp, env) 🡺 exp

Var references ;; x

eval(<var-ref> exp, env) 🡺 apply\_env(env, exp.var)

Compound Expressions

Define expressions ;; (define x (+ 5 6))

eval(<def-exp exp>, env) 🡺

val = eval(exp.val,env)

extend\_env(env, exp.var, val)

return undefined

If expressions ;; (if (> x 4) 5 6)

eval(<if-exp exp>, env) 🡺

test = eval(exp.test, env)

test ? eval(exp.then, env) : eval(exp.alt, env)

Literal expression ;; ‘abc ‘(1 2 (3 4))

eval(<lit-exp exp>, env) 🡺 exp.val

Procedures ;; (lambda (x) (\* x x))

eval(<proc-exp exp>, env) 🡺 make\_closure(exp.args, exp.body)

Procedure/Operator application ;; (+ 3 4) ;; ((lambda (x) (\* x x)) 3)

eval(<app-exp exp>, env) 🡺

args = [ eval(rand,env) for rand in exp.rands ]

proc = eval(exp.rator,env)

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.

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.

מודל ההצבה - כדי לחשב פרוצדורה נתונה (Closure) על פי רשימת ארגומנטים:

* מציבים ב body של הפרוצדורה את הארגומנטים, כל אחד מחליף 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)

הערה: 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

* + 1. מימוש האינטרפרטר

1. תשתית

הגדרת הערכים השונים: [L3-value.ts](https://github.com/bguppl/interpreters/blob/master/src/L3/L3-value.ts)

מימוש הסביבה: [L3-env.ts](https://github.com/bguppl/interpreters/blob/master/src/L3/L3-env.ts)

1. קוד האינטרפרטר

המסגרת הראשית:

[L3-eval.ts](https://github.com/bguppl/interpreters/blob/master/src/L3/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)

evalDefineExps

הרחבת הסביבה ע"פ ביטוי defineהראשון, 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](https://github.com/bguppl/interpreters/blob/master/src/L3/evalPrimitive.ts)

applyPrimitive

מימוש ב TS של כל אחד מהאופרטורים הפרימיטיביים ב 3L

חישוב הפעלת קלוז'ר:

פונקציה ראשית

[L3-eval.ts](https://github.com/bguppl/interpreters/blob/master/src/L3/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](https://github.com/bguppl/interpreters/blob/master/src/L3/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 (x)

(lambda (z)

(x z)

)

)

(lambda (w) (+ w z))

)

-->

(define z 3)

(lambda (z)

((lambda (w) (+ w z)) z)

)

בעיה: ההצבה מטשטשת את ההבדל בין z בפרוצדורה עם הפרמטר w, הקשור ל z בdefine , 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)

)

* + 1. Normal Order

ראינו כי בגישת Applicative Order מחשבים את הערך של כל האופרנדים לפני הפעלת הפרוצדורה.

ציינו כבר קודם בהערה, כי בגישת Normal Order לא מחשבים את הארגומנטים לפני הפעלת הפרוצדורה, אלא רק כאשר זה נדרש.

דוגמאות:

א.

)

(lambda (x) (+ x x))

(\* 2 3)

)

אפליקטיב: נחשב תחילה את (\* 2 3) ונציב

(+ 6 6)

נורמל: נציב (\* 2 3) 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 (\* 2 3) יבוצע פעמיים.

ב.

(

(lambda (x y z) (if x y z))

#t 3 (\ 30 3 2)

)

אפליקטיב: כל שלושת האופרנדים מחושבים מראש

(if #t 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 (\ 30 3 2)

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.

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.

שקילות:

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 ללא שגיאת זמן ריצה, אז חישובו ב normal order שקול. כלומר, סדר החישוב לא משנה במקרה זה (כאשר אין כמובן side-effects)

מימוש: [L3-normal.ts](https://github.com/bguppl/interpreters/blob/master/src/L3/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.
  1. מודל הסביבות
     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.

🡸 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 ניתו 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.

* + 1. הגדרת מודל הסביבות

1. מבנה נתונים

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

* נרחיב קונספט זה לכדי היררכיה של סביבות/פריימים (שהשורש שלה הוא הסביבה הגלובלית).
* בכל הפעלה של פרוצדורה, במקום לשכתב אותתה כמו במודל ההצבה, נגדיר סביבה/פריים חדש, שבו יוגדרו ה bindings של הפרמטרים של הפרוצדורה עם הערכים שנשלחו עבורם. סביבה זו תתווסף להיררכיה, כלומר תקושר לאחת הסביבות/הפריימים הקיימים.
* כאשר יש VarRef, האינטרפרטר יחפש את ערכו בהיררככית הסביבות/פריימים, מהסביבה/הפריים האחרון ולמעלה.

כלומר:

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

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) 🡺

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(env,make\_frame(proc.params, args)))**

דוגמאות:

א.

(define y 3)

(

(lambda (x) (+ x y))

(+ 5 2)

)

Y :3

GE

X : 7

E1

B1

P: x

B1: (+ x y)

ב.

(define x 1)

(

(lambda (f)

(

(lambda (x) (f x))

2

)

)

(lambda (y) (+ x y))

)

x :1

f :

E1

B1

P: f

B1: (  
 (lambda (x) (f x))

2

)

GE

P: y

B2: (+ x y)

x : 2

E2

B3

P: x

B3: (f x)

y : 2

E3

B3

Value: 4 (instead of 3)

האופן שבו בנינו את היררכיית הסביבות – חיבור הפריים החדש של הפעלת פרוצדורה לסביבה הנוכחית של זמן ההפעלה, כלומר לפריים האחרון – גרמה לכך, שהפרמטר x של הפרוצדורה (lambda (y) (+ x y)) התפרש כערך של x בזמן הפעלת הפרוצדורה ולא בזמן שהפרוצדורה נוצרה.

(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 יחובר לסביבה שהיתה בזמן שהיא נוצרה.

עדכון חוקי החישוב:

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)))

x :1

E1

GE

B1

f :

P: y

B2: (+ x y)

E2

B3

x : 2

P: x

B3: (f x)

E3

B3

y : 2

P: f

B1: (  
 (lambda (x) (f x))

2

)

'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

vars = vals in exp.bindings

cvals = [ eval(val,env) for val in vals]

eval\_sequence(exp.body, ext\_env(**env**, make\_frame(vars,cvals))

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)))

E1

GE

a : 1

E2

B2

b : 2

B1

הערה: השוואה למודל ה Activation Frame ב Java/C

* ההיררכיה ב Java/C היא שרשור של פריימים, כאן מדוובר 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)
  + 1. עדכון קוד האינטרפרטר עבור מודל הסביבות

תשתית:

מימוש מבנה הנתונים של הסביבות: [L4-env.ts](https://github.com/bguppl/interpreters/blob/master/src/L4/L4-env.ts)

עדכון הערכים (כעת מבנה ה Closure כולל שדה נוסף המציין את הסביבה שבה הוא הוגדר): [L4-value.ts](https://github.com/bguppl/interpreters/blob/master/src/L4/L4-value.ts)

האינטרפרטר: [L4-eval.ts](https://github.com/bguppl/interpreters/blob/master/src/L4/L4-eval.ts)

evalProc – הוספת הסביבה הנוכחית ל'לוז'ר המוחזר

applyClosure4 – מימוש חוק החישוב עבור הפעלת פרוצדורה (בניית פריים וחיבורו לסביבת הגדרתה)

evalLet4 – מימוש חוק החישוב עבור let (הגדרת פריים עם המשתתנים הלוקאליים וחיבורו לסביבה הנוכחית)

* + 1. Object Oriented Programing in L2

make-adder :

E1

GE

B1

a : 3

B1

P: a

B1: (lambda (x) (+ a x))

a3 :

P: x

B2: (+ a x)

a5 :

a : 5

E2

P: x

B2: (+ a x)

make-pair :

E2

GE

a : 3  
b: 4

B1

P: a,b

B1: (lambda (a b)

(lambda (sel) (sel a b)))

P: sel

B2: (sel a b)

p :

B1

E1

(let ((fact (lambda (n)

(if (=n 1)

1

(\* n (fact (- n 1))))))

(fact 2))

GE

E2

n : 2

B2

P: n

B2: (if (= n 1) 1

(\* n (fact (- n 1))))

B1

E1

fact :

(define fact

(lambda (n)

(if (= n 1)

1

(\* n (fact (- n 1))))))

(fact n)

empty

GE

E1

B1

n : 2

fact :

P: n

B1: (if (= n 1) 1 (\* n (fact (- n 1))))

(letrec ((fact (lambda (n)

(if (=n 1)

1

(\* n (fact (- n 1))))))

(fact 2))

GE

E2

n : 2

B2

P: n

B2: (if (= n 1) 1

(\* n (fact (- n 1))))

B1

E1

fact : <[n], [(if (= n 1) 1 (\* n (fact (- n 1)))]>

(define fact

(lambda (n)

(if (= n 1)

1

(\* n (fact (- n 1))))))

(fact n)

P: n

B1: (if (= n 1) 1 (\* n (fact (- n 1))))

GE

empty

E1

B1

n : 2

fact : <[n] [(if (= n 1) 1 (\* n (fact (- n 1))))]>

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

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

(even? 2)

GE

empty

E1

B1

n : 2

even? : <[n], [(if (= n 0) #t (odd? (- n 1)))]>

P: n

B1: (if (= n 0) #t (odd? (- n 1)))

odd? : <[n], [(if (= n 0) #f (even? (- n 1)))]>

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

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

(even? 2)

GE

P:

B1:

E1

B2

P:

B2:

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

(even? 2))

P:

B3:

P:

B2:

E1

B1

GE

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