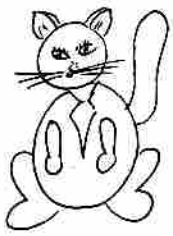
① 27.02.07
PLLAB



המרצה: אילן ilan_n@cs.huji.ac.il

pllab@cs.huji.ac.il

news group local.course.pllab.ta/stud

מבחן - אין

תרגילים - יש. בדרך אחת לשבוע. מגישים אלקטרונית. צריך לעשות register כדי להגיש תרגילים.

הגשה - קובץ קוד

קובץ בפיקוח

בראיתי

יש לקרוא את ההרצאות שמופיעות באתר.

תכנות פונקציונלי: זה משהו שונה מתשבתית ממה שלמדנו עד כה. בתכנות פונקציונלי יש ביטויים לכל אחד יש ערך. פחות מסתכלים על ביצוען. בשונה יש ערכים שמתחלפים.

השפה שנלמד איתה היא scheme. יש לינק להורדה באתר.

ב scheme יש 4 טיפוסים של מספרים: 3e4, $\frac{3}{4}$, 3.4, 34

טיפוסים בוליאניים - #t, #f

את הפעולות רושמים כ- prefix. לומר 4+5 לא 5+4.

scheme עושה DFS על העץ של הביטוי ומן פעם מתגלגל הרכבה בביטוי אטומי.

בכל מקום שכותבים בו ביטוי אטומי אפשר לרשום פונקציה כי יהיה לה ערך שאפשר לשערך וכך להמשיך הלאה.

כמו כן שפה שמרבדת את עצמה יש התניה שה סינטקס שלה הוא לה:

(if <exp> <exp> <exp>)

למשל (if (< 1 2) 4 17). במקרה זה הפלט הוא 4 ואילו 17 לא משוערך כלל.

ב scheme תמיד חייבים להגדיר את החלק של ה else כי אחרת תהיה בעיה לשערך.

יש גם פקודה שדומה ל- switch :

```
cond (((...)(...))
     ((...)(...))
     (else (...)) )
```

אפשר להגדיר קבועים (define my-pi 3.14)

באותו אופן מגדירים גם פונקציות (define (square x) (* x x))

אין פה שום טיפוסיות. אפשר לרשום גם (square 2) וגם (square 3/4)

בדוגמה שהגדרנו פונקציה נרלה לעשות רקורסיה

```
(define (sum_from_one n)
  (if (= n 1) 1
      (+ n (sum_from_one (- n 1)))))
```

איך כך מסתדר עם המודל שיצרנו? הוא משנה רק השרבר ממש ברגע לתשב אותו. כלומר כשהוא יקבל קלט ספציפי, אז לאותה פונקציה אינסופית אפשר הרבה יפרתה לעומק שצריך לפי הקלט.

② 6/3/07
Pllab

אפשר להגדיר מתפקש מה = עושה. לדוגמא

```
(define (= a b)
        (or a b)
)
```

אפשר להעביר פונקציות כפרמטר, פשוט מעבירים את השם של הפונקציה והפנים אפשר להשתמש בה כרגיל.

אפשר להגדיר פונקציה בלי לתת לה שם. (lambda (x)(* x x))
ואת זה אפשר להעביר לפונקציות. ההבדל בין זה לבין הגדרה באמצעות define זה כמו ההבדל בין (define pi 3.14)
לבין סתם הערך 3.14.

מערכות הדוק הנכונה להגדיר פונקציות הוא

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

כמו שאפשר להבל פונקציה אפשר גם להחזיר פונקציה.

```
(if (< y 2)
    (lambda (x) (+ x 1))
    (lambda (x) (+ x 3))
)
```

מה שמחזיר זאת פונקציה שמחכה לקבל הכניסה שלה מישהו שאין לו אותה.

אפשר להגדיר את הפונקציה וישר להפעיל אותה על מספר.

```
((lambda (x) (*xx)) 5 )
```

מבנה הנתונים הכי בסיסי שיש זה רשימה.

pair זה זוג של שני ערכים (•,•). מגדירים אותו ע"י construct: (cons 1 3) משמש לשים לאיבר הראשון משתמשים ב- car ובשביל האיבר השני משתמשים ב- cdr.

האיברים לא חייבים להיות מאותו סוג. בפרט יכול להיות זוג של זוגות או זוג של זוג ומספר.

אפשר לייצר מבנה נתונים של רשימה מקושרת ע"י זוגות שיוצאים אחד מתוך השני, ומסוף רשימה ריקה.

(cons 1 (cons 2 (cons 3 ()))

יש דרכים יותר נוחות להגדיר רשימה: (quote (1 2 3 4 5))

'(1 2 3 4 5)

(list 1 2 3 4 5)

ההבדל בין list לבין השניים הקודמים הוא ש list גם מבצע פעולות על הפרמטרים אבל quote משאיר אותם במו שהם. לא של

(quote (1 (+ 1 1) 3 4 5)) = (1 (+ 1 1) 3 4 5)

(list 1 (+ 1 1) 3 4 5) = (1 2 3 4 5)

פונקציה שצריך להכיר ברשימות.

reverse - הופך רשימה

append - משרשר רשימות (לא כמו שני זוגות, אלא הופך אותה לרשימה אחת שהיא השרשור)

map - מחזירה רשימת הערכים של הפעלת פונקציה על כל איברי הרשימה.

③ 13.3.07
PIlab

הגדרה אפמינוסטרטיבית: התפרסם באותה תאריך למבחן.
אין מבחנים לא להיכנס לפחות 5

ב scheme יש משהו שדומה למשתנים.
נניח שיש פונקציה מסוכרת:

```
(define complex_func (lambda (l) 5))
```

ויש פונקציה אחרת שמשתמשת ב-complex_func. אם היא
משתמשת בה הרבה נוכל לא יעזור אז הינו רוצים להגדיר משתנה
שמקבל את ערך הפונקציה ואז לבדוק אותו.
עושים binding באופן זמני לרשימת שמות

```
( let ( (x1 e1)
        (x2 e2)
        :
        (xn en)
      )
  e)
```

← הביטוי שמשערכים

איך משתמשים בזה?

```
(define list_proc
    (lambda (ls)
        (let ( (ans complex_func ls))
            )
        (cond (=0 ans)
              :
        )
    )
)
```

סדר הבילוי: - משערכים את n הביטויים
- משייכים אותם לשמות
- משערכים את הביטוי e

כה חשוב רשיש תלויות בין הביטויים.

בצורה דומה אפשר להגדיר פונקציות פנימיות שלא מוכרות מתוך ל- let שבו הן הוגדרו.

define זה להגדרות גלובליות! let זה לקישור מקומי.

יש מובן שלרוב אתה שם משלבים ואז מקשרים מיד. ואז למשל זה יהיה חוקי:

```
(let*
   ((x 5)
    (y x)   <- כי y מכיר את x בהגדרה שלו.
   )
   (+ x y)
)
```

לכן תמיד עדיף להשתמש ב- let*. למשל את הקוד הבא:

```
(define x 7)
(define y 8)
(let ((x y)
      (y x))   <- לא בשום מתחיל את x ו- y!
   (+ x y)
)
```

לאנונימי היה בשום צורה. לרשום בלי צורך let.

יש מציה עם פונקציות רקורסיביות כי ברגיסה השנייה לתוך הפונקציה הנסתר שלה כבר לא מוגדר בשבילה זה יש את letrec והוא מאפשר להגדיר פונקציות רקורסיביות ופונקציות שקוראות אחת לשנייה. סדר השיקרוק ב letrec הוא חלוי אימפלמנטציה אבל עדיף לא להשתמש בו אלא לפונקציות רקורסיביות.

20.3.07
Pllab

אפשר להגדיר רשימה של סמבולים - (line 2 point)'
וכך מתייחס לכל מה שבפנים בתור סימבול.
ואפשר גם ככה: (list 'line 2 'point) ואז הוא מתייחס
ל-2 כאל ערך נומרי ו- line ו- point הם סימבולים.

אפשר לבדוק אם משהו הוא סימבול ע"י symbol? והשוואה
ושוויון ע"י eqv?.

הפונקציה eval משערכת. אם נתנים לה סימבול היא משתמשת
בו במובן הרגיל. היא מוציאה למה הם מכוונים ומחזירה את
הערך. אם נתנים לה סתם משתנים רגילים אין היא
פשוט מחזירה אותם הביתה.

למשל כך אם כן אפשר בכל רגע לשנות פונקציות. אם יש
☆ (define funcA (lambda (ls)
(length ls)))

* (define myprog
'(define funcA (lambda (ls)
(* 2 (length ls)))))

☆ מגדיר פונקציה funcA. אבל * לא עושה כלום.
ברגע שנכתוב (eval my-prog) אז האמת תתבצע
הגדרה של funcA.

רקורסיה - יש שני דברים שאפשר להסתכל עליהם -
רקורסיה כמן ורקורסיה מקומי.
למשל חישוב עצרת
(define fact (lambda (n)
if (=n 0) 1
(* n (fact (-n 1))))))

סיבוכיות הזמן היא ליניארית ואפילו זה לא משהו מבחינת זיכרון. עד שהגיע לא מגיע עד n=0 היא לא יכולה לזכור את הביטוי אז היא כל הזמן הלה מתיק ביטוי בתוך בזיכרון.

אפשר לחשוב יעילות משהו בזמן זה רישום פיבונצ'י. בתרישום הרגיל ע"י הקשר הרקורסיבי $f(n)=f(n-1)+f(n-2)$ יש מספר אקספוננציאלי של קריאות.

בתוסר יעילות הזה ניתן לטפל ע"י הגדרת פונקציות עזר פנימיות.

הקורסיה זנב tail recursion

tail position - נקודה שבה הפונקציה מתלטה שהיא מחזירה ערך של פונקציה כלשהי. הספציפיקציה של sheme אומרת שהקומפיילר חייב לממש את זה ביעילות - ואז המחסנית לא מתפוצצת. בדיוק שמגיעים לחלק של ה... return b הוא פשוט שולח כל מה שנמצא בקופה במחסנית ומכניס רק את מה שרלוונטי ל- b.

שוויון כמו שכבר ראינו = לא עובד עבור משתנים מלאינוים. הפונקציות eqv? equal? ו- eq הן פרוימיטיביות - הן נותנות תשובה לפי טיפ משתנה. הפונקציות האלה הן יותר שקולות ואם הן מקבלות שני משתנים מסוגים שונים הן מחזירות false.

eqv? עובדת על bool, symbol, number, char, string, pair, list, empty list ו- function.
אבל על רשימות הנמוכים הוא מחזיר true רק אם הם הם ישמשו ממש מאותו מקום בזיכרון. equal? עובדת כפי שלכל אלה.

הטיפוסים קלת מורכבים לא ברור שהם הוא בהכרח שווים של ערכים. יכול להיות שפונקציא נכון להשוות כפוינטרים. למשל פונקציות אי אפשר להשוות לפי ערכים. זה פשוט מתי אפשרי. הדרך הנכונה להשוות פונקציות היא לפי פוינטרים.

ב - scheme כשמגדירים identifier לא מגדירים את הטיפוס שלו והוא יכול לקבל כל דבר. הדרך היחידה לבדוק טיפוס היא בזמן ריצה.
זה שכן נתקן זה מספר הפרמטרים - מוודאים שמספר הפרמטרים בזה לא תימה של הפונקציה.

בשפה שבה לכל ביטוי יש ערך בזמן קומפילציה אפשר לבדוק שרשימה היא הומוגנית - כל איבר שכל המשתנים בפנים מאותו סוג. ב - scheme זה מלמתי אפשרי. אבל הטיפוס הוא סתם list ולא list(int) למשל.
אבל אפשר מאמץ לבדוקה הכל. זה יתרון גדול. אבל החיסרון הוא שאין שום בדיקה.

⑥ 27.3.07
PllAb

צריך ותחול לראות את הדברים המתלחמים יתר של שפות פונקציונליות.

אך צריכים הפונקציות קיבלו מידע פרימיטיבי והתצירו מידע.
אינו יום פונקציות שמקבלות ומתצירות פונקציות אבל ברור לה היה שלב עכב לפונקציות שמעבדות מידע.

יש אלגוריתמים שמקבלים פונקציות כפרמטר. למשל sort מקבלת מערך ופונקציית השוואה. אבל sort הינו מסדר גבוה. הינו לאו קשורה לשום data ספציפי.

מה מאפשר לוגיקור אל גבוה של פונקציות. ואיך אפשר להתמקד בגלגוריתמים בלי שיהיה ארכסת לנו איך פונקציות ספציפיות ממומשות.

למשל אם יש פונקציה שמרכיבה פונקציות ופונקציה שמתצירה נגזרת איך יש אותנו יכולים לחשב נגזרת מכל סדר.

streams

פונקציות קבלה מנסות לשערך את האוטומטים שהן וכל לפעול. אבל יש בעיה אם פונקציה מנסה לקבל את עצמה.

ביטוי תסר פרמטרים נקרא promise - (lambda () 2)
ושמשערים אותו מקבלים 2.

stream זה רגל של המטמות.
המטמה בה משהו שלא תפס ציפון, זה בעצם לאו ביים עד שלא מבקשים אותה. איך stream יכול להיות אינסופי.

5.6.07
PLLAB

הודעה: תרגיל 5 נדחה בכמה ימים. בהצלחה!

* * * * *

היום נדבר על תכנות לא פונקציונלי. מסתבר שאי אפשר לעשות הכל בתכנות פונקציונלי אז יש ב-scheme אופציה לתכנת גם לא פונקציונלית.

אנחנו נדבר על פקודות שיש להם תופעות לוואי, כלומר הן משפיעות על סביבת העבודה.
הדוגמה הראשית למשהו כזה היא פקודות קלט/פלט. למשל, יש שתי פונקציות שפולטות למסך – write ו-display. סה"כ הן עושות את אותו הדבר אבל יש ביניהן כמה הבדלים קטנים. יש גם פונקציות שקולטות מהמשתמש תווים בודדים או מחרוזות שלמות.
Begin מאפשר לנו להריץ רצף של פקודות אחת אחרי השנייה. זה נוח בשביל דיבאג אן נניח רוצים להדפיס פרמטרים שמועברים לפונקציה.
אפשר לעשות השמה. הסינטקס הוא (set! Identifier expression). ההבדל בין set! ל-define הוא ש-define מקצה תא בשיכרון ושם בו ערך ואילו set! מחפש את התא בזיכרון שמוקצה ל-identifier ושם בו ערך. לכן אי אפשר לעשות השמה לתוך משתנה שלא עשו לו define קודם. יש פקודות דומות עבור זוגות – set-car! ו-set-cdr!

Scheme יודעת לזעות תאים שאין אליהם מצביע ולמחוק אותם. אז אנחנו לא צריכים לדאוג לדליפות זיכרון.

החלק האחרון של הקורס – גרפיקה. יש חבילת גרפיקה שמאפשרת לעשות כל מיני דברים.
באתר יש דוגמאות קוד. מומלץ להסתכל על העץ הפרקטלי.

סוף