אמיר ארבל 302726310, יותם דיקשטיין 201212198

שאלה 1:

1. בשתי האפשרויות:

נשים לב שכבר בשלב ה-define ה-preprocessor החליף את ה-let באופציה הבאה:

(define normal? (lambda() ((lambda (e) (display 'normal)) (display 'not-))))

a. אפליקטיבי:

- app-eval[(normal?)]

הפרוצדורה normal? היא פרוצדורה ללא פרמטרים, לכן לא מתבצעת החלפה, והביטוי היחיד בגוף הפרוצדורה עובר להערכה אפליקטיבית והערך שיוצא מוחזר כתשובה:

- app-eval[((lambda (e) (display 'normal)) (display 'not-))]

הביטוי (display 'not-) מחושב ב-app-eval (ומדפיס not). והערך שיוצא בו מוחלף בכל המקומות בהם e נמצא בגוף הפרוצדורה (אין מקומות כאלה, לכן בפועל אין השפעה).
לאחר מכן מחושב display normal (שיחזיר void וידפיס תו"כ normal) ותשובה זו מועברת כתשובה ל-app-eval.

b. נורמלי:

2. normal-eval[(normal?)]

הפרוצדורה normal? היא פרוצדורה ללא פרמטרים, לכן לא מתבצעת החלפה, והביטוי היחיד בגוף הפרוצדורה עובר להערכה נורמליתו והערך שיוצא מוחזר כתשובה:

- normal-eval[((lambda (e) (display 'normal)) (display 'not-))]

הביטוי (display 'not-) עושים לביטוי שבתוך הלמבדא החלפה של כל המופעים של המשתנה e בביטוי (display 'not-).
לאחר מכן מחושב (בשיטת נורמל) הערך של (display normal), ומכיוון ש-display הוא פרוצדורה פרמיטיבית, אז ערכה מחושב והפרוצדורה מחזירה את התשובה (ומדפיסה normal).ותשובה זו מועברת כתשובה ל-normal-eval.

2. לא ניתן לכתוב פונקציה שמדפיסה applicative אם ההערכה אפליקטיבית, ומדפיסה not-applicative אם ההערכה נורמלית, מכיוון שכל פונקציה כזאת צריכה לכלול פקודה של display 'not או display 'not-applicative', שאם היא מודפסת בנורמל, אז היא בוודאות מוערכת ומודפסת באפליקטיב.

3. שתי דרכי האבלואציה, במידה ושתיהן מצליחות, מחזירות אותו ערך, לכן לא ניתן לבנות פרוצדורה שמחזירה ערכים שונים במידה לפי סוג האבלואציה.

שאלה 3:

חלק ראשון - a

For every: type environment _Tenv,
variables _v1, _v2, ..., _vn, n>=0,
expressions _e1, _e2, ..., _en,
expressions _b1, _b2, ..., _bm, m>=1, and
type expressions _S1, ...,_Sn, _U1, ...,_Um :
If _Tenv {_b1:_U1, …,_bm:_Um}|-_e1:_S1,
_Tenv {_b1:_U1, …,_bm:_Um}|-_e2:_S2,
_Tenv {_b1:_U1, …,_bm:_Um}|-_e3:_S3,
...,
_Tenv {_b1:_U1, …,_bm:_Um}|-_en:_Sn,

_Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn, _b1:_U1, …,_bm:_Um} |-_b1:_U1,
_Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn, _b1:_U1, …,_bm:_Um} |-_b2:_U2,
...,
_Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn, _b1:_U1, …,_bm:_Um} |-_bm:_Um
Then
_Tenv |- (let ((_v1 _e1) (_v2 _e2) ... (_vn _en)) _b1 _b2 ... _bm):_Um

חלק ראשון - b

ננתח את:

(letrec ((foo (lambda (f) (goo foo f))) (goo (lambda (foo f) (lambda()(foo f))))) (foo (lambda (x) x)))

לפני כן, נעשה renaming למשתנים לפי הסביבה שלהם.

(letrec ((foo (lambda (f) (goo foo f))) (goo (lambda (**foo1 f1**) (lambda()(**foo1 f1**))))) (foo (lambda (x) x)))

נתחיל עם הגוף בלי התייחסות להגדרות ב-letrec:

{x:T1} |- x:T1 החוק למשתנים
{x:T1} |- (lambda (x) x): [T1->T1] החוק לפרוצדורות
_Tenv {} |- foo: [[T1->T1] -> T2] החוק לפרוצדורות

כעת ננתח את החלק של goo ב-letrec:

{f1:T3} |-f1:T3 החוק למשתנים
{foo1:T5}|-foo1:T5
{f1:T3, foo1:[T3->T4]} |-foo1:[T3->T4] החוק לפרוצדורות
{f1:T3, foo1:[T3->T4]} |-(foo1 f1):T4 החוק להפעלת פרוצדורה
{f1:T3, foo1:[T3->T4]} |-(lambda()(**foo1 f1**)): [E->T4] החוק לפרוצדורות

{} |- (lambda (foo1 f1) (lambda()(foo1 f1)): [[T3->T4]*T3 -> [E->T4]] החוק לפרוצדורות
{goo:T10} |- goo:T10
{goo:[[T3->T4]*T3 -> [E->T4]]} |- goo:[[T3->T4]*T3 -> [E->T4]] מקרה פרטי של החוק למשתנים
{} |- goo:[[T3->T4]*T3 -> [E->T4]] החוק עבור letrec.

כעת נסתכל על החלק של foo:

{f:T6} |- f:T6 החוק למשתנים
{foo:T7} |- foo:T7 החוק למשתנים
{f:T3} |- f:T3 מקרה פרטי של הצהרה קודמת
{foo:[T3->T4]} |- foo:[T3->T4] מקרה פרטי של הצהרה קודמת
{f:T3, foo:[T3->T4]} |- (goo foo f):[E->T4] החוק להפעלת פרוצדורות
{foo:[T3->T4]} |- (lambda (f) (goo foo f)):[T3->[E->T4]] החוק לפרוצדורות

נשים לב שהיינו רוצים לעשות substitution כדי ש- $T4 = [E -> T4]$ אך זהו substitution שאינו חוקי. לכן הביטוי הזה הוא חסר טיפוס.

חלק שני - a

define עם תמיכה ברקורסיה (כשזה בלי תמיכה ברקורסיה, עובדים בלי ההנחה בהרכבה):

For every: type environment _Tenv, recursive-definition expression (define _f _e) and type expression _S:
If _Tenv◦{_f:_S } |- _e:_S,
Then _Tenv |- (define _f _e):Void, and
{ } |- _f:_S

letrec:

For every: type environment _Tenv,
variables _v1, _v2, ..., _vn, n>=0,
expressions _e1, _e2, ..., _en,
expressions _b1, _b2, ..., _bm, m>=1, and
type expressions _S1, ...,_Sn, _U1, ...,_Um :
If _Tenv {_b1:_U1, …,_bm:_Um}|-_e1:_S1,
_Tenv {_b1:_U1, …,_bm:_Um}|-_e2:_S2,
_Tenv {_b1:_U1, …,_bm:_Um}|-_e3:_S3,
...,
_Tenv {_b1:_U1, …,_bm:_Um}|-_en:_Sn,

_Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn, _b1:_U1, …,_bm:_Um} |-_b1:_U1,
_Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn, _b1:_U1, …,_bm:_Um} |-_b2:_U2,
...,
_Tenv {_v1:_S1, _v2:_S2, ..., _vn:_Sn, _b1:_U1, …,_bm:_Um} |-_bm:_Um
Then
_Tenv |- (let ((_v1 _e1) (_v2 _e2) ... (_vn _en)) _b1 _b2 ... _bm):_Um

חלק שני - b

ביטוי ראשון:

(define f (lambda (n) (letrec ((g (lambda (n) (if (< n 0.001) n (g (/ n 2)))))) g)))

(define f (lambda (n) (letrec ((g (lambda (n) (if (< n 0.001) n (g (/ n 2)))))) g)))	T1

אמיר ארבל 302726310, יותם דיקשטיין 201212198

f	Tf
(lambda (n) (letrec ((g (lambda (n) (if (< n 0.001) n (g (/ n 2)))))) g))	T3
(letrec ((g (lambda (n) (if (< n 0.001) n (g (/ n 2)))))) g)	T4
(letrec ((g (lambda (n) (if (< n 0.001) n (g (/ n 2))))))	T5
g	Tg
(lambda (n) (if (< n 0.001) n (g (/ n 2))))	T7
(if (< n 0.001) n (g (/ n 2)))	T8
(< n 0.001)	T9
n	Tn
0.001	Tnum0.001
(/ n 2)	T10
<	T<
/	T/
2	Tnum2
(g (/ n 2))	T11

T/:[Tn*Tnum2->T10]
T<:[Tn*Tnum001->T9]
Tg:[T10->T11]
T8:Tn union T11 (since we decided that in an if condition both expressions should be from the same type then wlog. T8:Tn)
T7:[Tn->T8]
Tg:T7 (from letrec rule)
T3:[Tn->Tg]
Tf:T3 (from define rule)

from the rules for numbers, primitive procedures, and define:
Tnum2:N
Tnum0.001:N

T/:[N*N->N]
T<:[N*N->B]
T1:void

from this and the first two equations we get :
Tn:N
T10:N
T9:B

T8:N, T11:N
Tg:[N->N]
T7:[N->N]
T3:[N->[N->N]]
Tf:[N->[N->N]]

ביטוי שני:

((f 3) 4)

f	Tf
3	Tnum3
4	Tnum4
(f 3)	T12
((f 3) 4)	T13

Tf:[Tnum3->T12]
T12:[Tnum4->T13]

from define before we get:
Tf:[N->[N->N]]
thus:
Tnum3:N (is OK since this we also get from number rule)
T12:[N->N]
Tnum4:N (is OK since this we also get from number rule)
T13:N

מכאן שלאחר הביטוי הראשון נחזיר טיפוס void ולאחר הביטוי השני נחזיר טיפוס Number.

חלק שלישי:

1. שלוש סיבות לכישלון:

a. יתכן כי נסיק בסביבה שלנו טיפוסים בצורה מעגלית $\{T_1 : [T_1 \to T_2]\}$ כמו למשל: $(lambda\ (x)\ (x\ x))$.
b. האלגוריתם יכשל כי נגיע לשני שיוויונים סותרים בסביבה $\{T_1 = N,\ T_1 = B\}$, כמו למשל: $(lambda\ (x)\ (x\ 1)\ (+\ x\ 1))$ - בה פעם אחת ה-x צ"ל Number ופעם אחרת Proc.
c. האלגוריתם יכשל כי לא ניתן להגיע לטיפוס בלי הנחות ב-Tenv. למשל $(+\ x\ 1)$.

2. אנו צריכים חוק מיוחד ל-define כדי שנוכל להסיק את הטיפוס של **המשתנה שהוגדר ב-define**. למשל, אם קיבלנו את הביטוי $(define\ x\ 1)\ (+\ x\ 1)$, לא נוכל להסיק ש-x הוא מספר בלי הכלל ל-define, ולכן הביטוי השני לא יקבל טיפוס.

אמיר ארבל 302726310, יותם דיקשטיין 201212198

שאלה 4:

1. א.
?(How many function applications are computed when evaluating (c1-n 3 add1
למעשה, אחת, - כאשר הsubstition ריק והתוצאה היא רק קלוזור חדש.
ב.
How many when evaluating ((c1-n 3 add1) 1)? Explain by showing the main steps of the applicative-eval steps of this expression.
נראה פיתוח של ההרצה: (נתייחס ל

1.ae[((c1-n 3 add1) 1)]	2. ae[3] -> 3	3. ae[add1]= closure<primitive>
4.FA(1) c1-n [n=3, f=add1]	5. ae[(lambda (x) (if (= n 1) (f x) ((c1-n (- n 1) f) (f x))))] =create-closure < (if (= 3 1) (add1 x) ((c1-n (- 3 1) add1) (add1 x))))>	6.FA(2) (c1-n 3 add1) [x=1] = (if (= 3 1) (add1 1) ((c1-n (- 3 1) add1) (add1 1))))
7. FA(3) [substitute into the procedure...] (= 3 1) = #f	8.ae[((c1-n (- 3 1) add1) (add1 1)))] = closure<..>	9.FA(4) [x1=3] [x2=1] (- x1 x2) =2
10. ae[add1]=closure<primitive>	11.FA(5) c1-n [n=2, f=add1] = closure<..>	12. [like step 5, but n=2]
13. FA(6) [x1=1] (add1 x1) = 2	14. FA(7) [x=2] (if (= 2 1) (add1 2) ((c1-n (- 2 1) add1) (add1 2))	15. FA(8) (= 2 1) = #f
16. FA(9) (*Like 13 with x1=2) = 3	17. FA(10) (*like 9 with x1=2, x2=1*) = 1	18. FA(11) (* like 4,5 with n=1 f=add1)
19. FA(12) (*like 13 with x1=2) =3	20. FA(13) (*like 15 with x1=1, x2=1) = #t	21. FA(14) (* like 13 with x1= 3) = 4

סה"כ 14 function applications.

2. א. 38 פעמים: c2-n*10, =*10, -*9, compose*9
ב. 13 פעמים: c2-n*3, =*3, -*2, add1*3, compose*2
3. כן, מכיוון שח-c2 מחושב ל-n איברים פעם בודדת בזמן ריצה ולמעשה כל הפונקציות "מחכות" לקבל ערכים (eager). לעומתה ח-c1-n מחושבת בזמן ריצה ברגע שמגיעים אליה בסדר פעולות, ולכן פעולתה מבוצעת בזמן ארוך יותר.
4. א. מבוצעות: 1*/, compose*2, -*1, even?*2, =*3, (comp1-n n)*3, comp1-n*3
ובסה"כ: 15.
בוצע =, even?, - (או \ אם צעד זוגי) ואז באופן רקורסיבי ח-comp1-n והפעלתו, ורק לאחר מכן

compose (בצעד זוגי יבוצע comp-1n, אחריו compose ורק אז הפעלת comp-1n).

ב. לכל צעד (זוגי או אי זוגי), מבוצעות כ-6 פעולות, ול-n כלשהו יבוצעו כ $6 * upper-value\left[log_2 n\right] + 3$ פעולות (3 פעולות מבוצעות בסוף "הדרך").

5. א. מבוצעות:

compose*3, cn-1*1, cn/2*1,/*1,-*1,comp2-n*2,even?*1, =*3 ובסה"כ: 12

כאשר compose מופעלות בסוף ביצוע תוכן שאר הפעולות.

ב. נסתכל באופן כללי על הפעלה על מספר n=2^k-1 כלשהו ופונקציה f:

1.ae[((comp2-n n) f)]	2.ae[n] -> n=2^k-1	3.FA (1) [n=n=2^k-1] comp2-n
4. ae[(cond ((= n 1) (lambda (f) f)) ((even? n) (let ((cn/2 (comp2-n (/ n 2)))) (lambda (f) (cn/2 (compose f f))))) (else (let ((cn-1 (comp2-n (- n 1)))) (lambda (f) (compose f (cn-1 f))))))]	5. FA(2), FA(3) - check conditions - ae[(= n 1)] -> ae[n] =n , ae[1] = 1 , n<> 1-> #f ae[(even? n)] -> ae[n] =n = 2^k-1 , -> #f	6. ae[(else (let ((cn-1 (comp2-n (- n 1)))) (lambda (f) (compose f (cn-1 f)))))]
7. ae[(let ((cn-1 (comp2-n (- n 1)))) (lambda (f) (compose f (cn-1 f))))]	8. FA(4) ae[(- n 1)] -> ae[n] = n, ae[1] = 1 -> n-1 = 2^k-2	9. FA(5) ae[(comp2-n n-1)]
10. ae[(cond ((= n-1 1) (lambda (f) f)) ((even? n-1) (let ((cn/2 (comp2-n (/ n-1 2)))) (lambda (f) (cn/2 (compose f f))))) (else (let ((cn-1 (comp2-n (- n-1 1)))) (lambda (f) (compose f (cn-1 f))))))]	11. FA(6,7) Like step 5, even? -> #t	12. ae[(let ((cn/2 (comp2-n (/ n-1 2)))) (lambda (f) (cn/2 (compose f f))))]
13. FA(8) ae[(/ n-1 2)] -> (n-1)/2 = 2^(k-1) -1	14 FA(9) .ae[(comp2-n 2^(k-1) -1)]	15. [recursively continue until n=1 - notice we apply 4 actions per “round”.] Two extra actions (compose and applying cn/2 or cn-1) are held in a function. - Total FA(1 + 4* (2k-1) + 2* (2k-1)) = FA(1 + 6 * (2k-1))=FA(1+t)

18. retroactively apply compose and cn/2, cn-1 functions: cn-1 happens before composition, so it “added” to the list of functions composed together, while cn/2 happens after composer, meaning the carried function is “twice as strong” - every succeeding function is using the composed function as a base function.	17. ae[(lambda (f) f)]	16.FA(2+t) ae[(= 1 1)] = #t
	20. substitute add1 into the composed function. Since we aren’t applying this function with a number, this is just one FA.	19. {at this point, we have a very long composed function waiting for a function as a parameter, that will create a new function}

נשים לב אם כן שביצענו, בתלות בגודל ו"זוגיות" n, כ- $O(log_2(n))$ פעולות.

6. הפרוצדורה החוזרת היא מסוג [[T->T]->[T->T]]. מה שלמעשה חוזר זה "הכנה" לפונקציה שתורכב על עצמה n פעמים - כאשר כלל הקלוז'רים הדרושים מחכים רק להצבת הפונקציה במקום המתאים- כלומר מוכנים ב"זמן קומפילציה" ופועלת מהר יחסית בזמן ריצה.

אמיר ארבל 302726310, יותם דיקשטיין 201212198

שאלה 5:

חלק שלישי:

ה-BNF של הביטויים החדשים:

<human-poly>::'(' 'p' <monome-exp>* <number> '')'
<monome-exp>:: <number> 'x' <positive-integer> '+'
<positive-integer>:: (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)(0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*

חלק חמישי:

1. צורת ה-evaluation של Ipoly היא אפליקטיבית, כלומר ערכים מחושבים מראש. לדוגמה, אם נכניס $(plus\ (plus\ (p'\ 0\ (1))\ (p'\ 0\ (1))\ (p'\ 0\ (2)))$ אז הפלוס הפנימי יחושב קודם. יחד עם זאת, נשים לב שמכיוון שאנו מתייחסים לכל הפרוצדורות שהגדרנו בקוד כפרוצדורות פרמיטיביות, אז בכל מקרה הערכים הפנימיים היו מחושבים קודם (כי גם הערכה נורמלית מחשבת את הערכים כשהם מוכנסים כפרמטרים לפרוצדורות נורמליות).
2. הערכים המוחזרים מ-Ipoly יכולים להיות poly (בהפעלת חיבור, כפל, נגזרת או בהנתן פולינום), מספר (בהפעלת apply או בהנתן מספר), או פרוצדורה $[Symbol -> Symbol \cup Poly \cup Number]$ (כשמפעילים define. ה-Symbol שמוחזר הוא ה-'none מה-env הריק).
3. מכיוון ששפת הפולינומים היא שפה רגולרית (זה לא קורס באוטומטים אז לא נביא הוכחה פורמלית, אבל נשים לב שהזיכרון שנדרש לבדיקה אם ביטוי כנ"ל הוא פולינום חוקי הוא חסום), אז בפרט היא ניתנת לתיאור ע"י BNF. כמו כן, כדי לתמוך בשפה שתוארה יש לעדכן את ה-concrete syntax בלבד (תיאור האובייקט נשאר אותו תיאור).