Q1a

```
(define g 1)
(lambda (a b c)
 (if (eq? [b: 0  1] [g: free])
     ((lambda (c)
        (cons [b: 1  1] [c: 0  0]))
      [a: 0  0])
     [b: 0  1]))
```

Q1b

```
(let ((z 1))               ; 1
 (((lambda (x)             ; 2
     (lambda (z) (x z)))   ; 3
   (lambda (w) z))         ; 4
  2))
```

Q1c

הרעיון הכללי: ע"י שימוש בכתובות לקסיקליות אנחנו יודעים לזהות באופן מדויק את כל המופעים של המשתנים שזקוקים להחלפה. כך שגם במקרה בו יש משתנה באותו השם, נדע להחליף רק את המופע שמתייחס לארגומנט, ולא מופעים אחרים. יש לשים לב כי כאשר סורקים את גוף הפונקציה על מנת להחליף את המשתנים, ונתקלים ב-contour-ים חדשים יש לשים לב שמונה העומק עולה בהתאמה.

כאשר מתבצעת החלפה, אם יש משתנה גלובלי בקטע **המחליף**, אזי משאירים אותו כמו שהוא.

במידה ויש משתנה שאינו גלובלי יש לעדכן את הכתובת הלקסיקלית (כמו בתשובה Q1b) - לשנות את העומק בהתאם לעומק המשתנה שמחליפים - לדוגמא:

```
(let ((z (lambda (x) (* x x))))
    (((lambda (x) (lambda (z) ([x : 1 0] [z : 0 0])))) ; 1
     (lambda (w) ([z : 1 0] [w : 0 0])))               ; 2
     2))
-->
(let ((z (lambda (x) (* x x))))
    ((lambda (z) ((lambda (w) ([z : 2 0] [w : 0 0])) z))
     2))
```

משנים רק את המשתנים שהעומק שלהם גדול מ-0

שגיאות נפוצות:

- תיאור הבעיה מתייחס רק לתפיסת משתנים חופשיים, למרות שבסעיף הקודם ראינו שתיתכן תפיסה גם למשתנים קשורים.
- בלבול בין מודל ההחלפה למודל הסביבות. למשל, הוספת מבנה נתונים וחיפוש בו.
- אי-עדכון הכתובת הלקסיקלי של משתנים קשורים בקטע המחליף.

Q1d

ע"י שימוש בכתובת לקסיקליות במודל הסביבות, בביצוע lookup של משתנה בסביבה, אנו יודעים בדיוק כמה מסגרות צריך לחזור אחורה בסביבה, ומה ההסט של המשתנה בתוך המסגרת שלו. אם נממש את הסביבות כמערך של מערכים, נוכל לבצע גישה ישירה למשתנה, ולא לחפש אותו מסגרת-מסגרת בסביבה.

שגיאות נפוצות:

- ביצוע איטרציה על הסביבות. אמנם זה חוסך חיפוש בתוך כל מסגרת, אבל זו עדיין לא גישה ישירה.
- בניית מפה (מילון) שהמפתח הוא כתובת לקסיקלית. כיוון שהכתובות הן יחסיות, בהחלט ייתכן שלשני משתנים שונים תהיה אותה כתובת לקסיקלית, מה שיוצר התנגשות.
- התעלמות מגישה ישירה למשתנה בתוך מסגרת, אלא גישה ישירה רק למסגרת עצמה.

Q2

2.1.1 `{x : T1, y : T2, f : [T2 → T1]} |- (f y) : T1`

`True`

2.1.2 `{f : [T1 → T2], g : [T1 → T2], a : T1} |- (f (g a)) : T2`

`False - (g a): T2 and f expects T1 as a param`

2.2

```
(L5
  (define f (lambda (n) (* n n)))
  (define g (lambda (h) (lambda (p) (h (h p)))))
  ((g f) 2))
```

Expression	Variable	Type
1. (L5 …)	T0	Program
2. (define f …)	T1	Def-exp
3. (lambda (n) (* n n))	T2	Proc-exp
4. (* n n)	T3	App-exp
5. *	T*	Prim-op
6. n	Tn	VarRef
7. (define g …)	T4	Def-exp
8. (lambda (h) (lambda (p) …))	T5	Proc-exp
9. (lambda (p) …)	T6	Proc-exp
10. (h (h p))	T7	App-exp
11. (h p)	T8	App-exp
12. h	Th	VarRef
13. p	Tp	VarRef
14. ((g f) 2)	T9	App-exp
15. (g f)	T10	App-exp
16. 2	Tnum2	Num-exp
17. g	Tg	VarRef
18. f	Tf	VarRef

```
Expression                          Equation
==============================      ==========================
   1. (L5 …)                        T0 = T9
   2. (define f …)                  T1 = void
                                    Tf = T2
   3. (lambda (n) (* n n))          T2 = [Tn -> T3]
   4. (* n n)                       T* = [Tn * Tn -> T3]
   5. *                             T* = [Number * Number ->
      Number]
   6. (define g …)                  T4 = void
                                    Tg = T5
   7. (lambda (h) (lambda (p) …))       T5 = [Th -> T6]
   8. (lambda (p) …)                T6 = [Tp -> T7]
   9. (h (h p))                     Th = [T8 -> T7]
   10.  (h p)                       Th = [Tp -> T8]
   11.  ((g f) 2)                   T10 = [Tnum2 -> T9]
   12.  (g f)                       Tg = [Tf -> T10]
   13.  2                           Tnum2 = Number
```

כתבו את ה-type הנגזר עבור g בתוכנית:

```
On the basis of the equations, we can solve for Tg:

                     Tg = [Tf -> T10]
                     Tg = T5

Through successive unification:

                     Tg = [Tf -> T10]
Tf = T2              Tg = [T2 -> T10]
T2 = [Tn -> T3]      Tg = [[Tn -> T3] -> T10]
T* = [Tn * Tn -> T3] = [Number * Number -> Number]
Tn = Number
T3 = Number          Tg = [[Number -> Number] -> T10]
T10 = [Tnum2 -> T9]  Tg = [[Number -> Number] -> [Tnum2 -> T9]]
Tnum2 = Number       Tg = [[Number -> Number] -> [Number -> T9]]

                     Tg = T5
T5 = [Th -> T6]      Tg = [Th -> T6]
Th = [T8 -> T7]      Tg = [[T8 -> T7] -> T6]
T6 = [Tp -> T7]      Tg = [[Number -> Number] -> [Tp -> T7]]
T7 = Number          Tg = [[Number -> Number] -> [Number -> Number]]

It is sufficient to directly provide the answer:
```

Tg = [[Number -> Number] -> [Number -> Number]]

Q3a

```
;; Signature: add-at-end$(lst a c)
;; Type: [List(T1) * T1 * [List(T1) -> T2] -> T2]
;; Purpose: append element a at end of list lst
;; Tests: (add-at-end$ '(1 2 3) 4 id) => '(1 2 3 4)
(define add-at-end$
  (lambda (lst a c)
    (if (empty? lst)
        (c (list a))
        (add-at-end$ (cdr lst) a
                     (lambda (res) (c (cons (car lst) res)))))))
```

Proof:

Base case:

a-e[(add-at-end$ ‘() a c)] = (c (list a)) = (c (add-at-end ‘() a))

Induction Hypothesis:

(add-at-end lst’ a c) = (c (add-at-end lst’ a)) where lst’ is a list of length less than some n.

Induction Step:

```
a-e[ (add-at-end$ lst a c) ]
= a-e[ (add-at-end$ (cdr lst) a
         (lambda (res) (c (cons (car lst) res)))) ]
=IH= a-e[ ((lambda (res) (c (cons (car lst) res)))
           (add-at-end (cdr lst) a)) ]
= a-e[ (c (cons (car lst) (add-at-end (cdr lst) a))) ]
= a-e[ (c (add-at-end lst a)) ] QED
```

Q3b

```
;; Signature: tree-reduce$(reducer$, init, tree, cont)
;; Type: [[T2 * T1 * [T2 -> T3] -> T3]
;;        * T2
;;        * Tree<T1>
;;        * [T2 -> T3]
;;        -> T3]
(define tree-reduce$
  (lambda (reducer$ init tree cont)
    (if (empty? tree)
        (cont init)
        (reducer$ init (tree->data tree)
                  (lambda (root)
                    (tree-reduce$ reducer$ root (tree->left tree)
                                  (lambda (left)
                                    (tree-reduce$ reducer$
                                                  left
                                                  (tree->right tree)
                                                  cont)))))))
```

Q4a

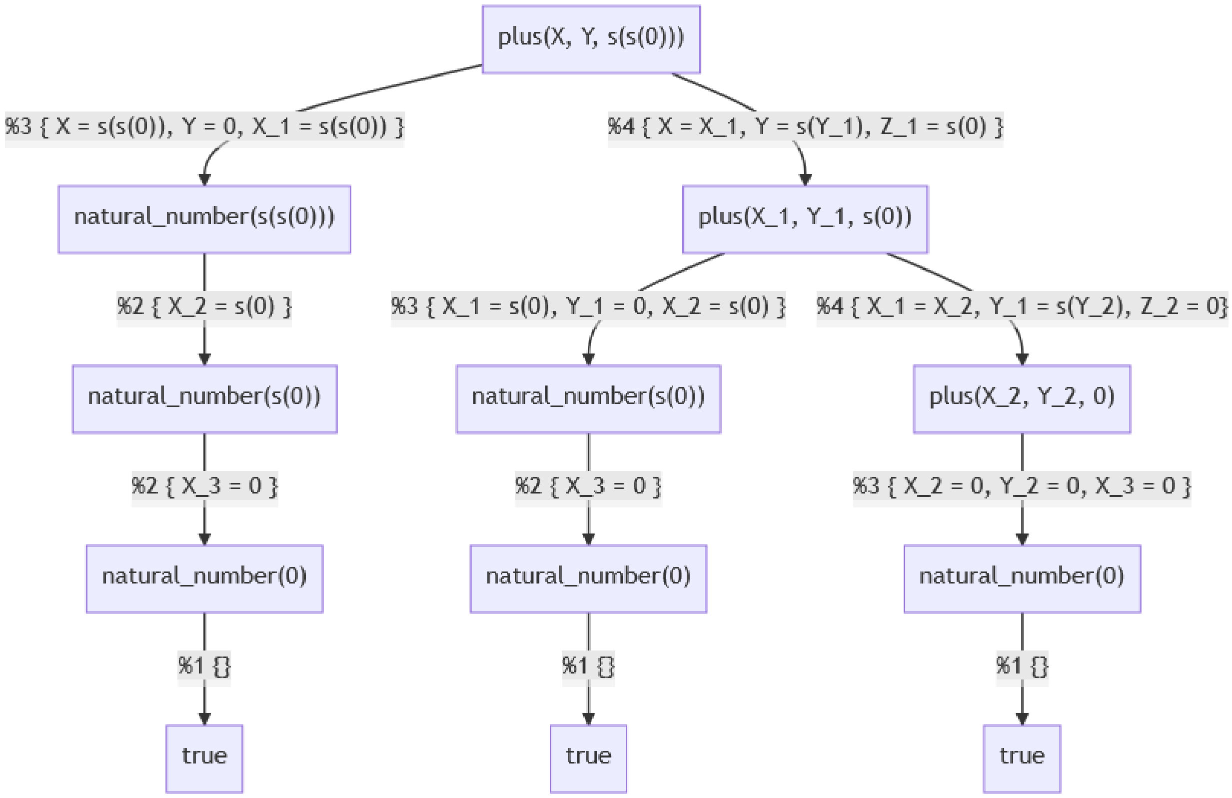
```
% Signature: take(List, N, Sublist)/3
% Purpose: Sublist is the first N elements from List

Here are three possible answers to this question:
% Version 1:
take([], _, []).                    %1
take([_|_], 0, []).                 %2
take([X|Xs], s(N), [X|Ys]) :-       %3
     take(Xs, N, Ys).

% Version 2:
take([], s(_), []).                 %1
take(_, 0, []).                     %2
take([X|Xs], s(N), [X|Ys]) :-       %3
     take(Xs, N, Ys).

% Version 3:
take([], _, []).                    %1
take(_, 0, []).                     %2
take([X|Xs], s(N), [X|Ys]) :-       %3
     take(Xs, N, Ys).
```

Version 3 gets full grade, but it has a problem that versions 1 and 2 do not have: it returns two answers for certain queries, for example: “?- take([1, 2, 3], s(s(s(0))), X).”.
This is avoided in Versions 1 and 2 by making sure that the rules %1 and %2 are disjoint.

Q4b

```
% Signature: pad(List, N, Padded)/3
% Purpose: Padded is List padded with *s to reach length N
pad(List, 0, List).
pad([], s(N), [*|Rest]) :-
     pad([], N, Rest).
pad([X|Xs], s(N), [X|Padded]) :-
     pad(Xs, N, Padded).
```

Q4c



plus(X, Y, s(s(0)))
%3 { X = s(s(0)), Y = 0, X_1 = s(s(0)) }
%4 { X = X_1, Y = s(Y_1), Z_1 = s(0) }
natural_number(s(s(0)))
plus(X_1, Y_1, s(0))
%2 { X_2 = s(0) }
%3 { X_1 = s(0), Y_1 = 0, X_2 = s(0) }
%4 { X_1 = X_2, Y_1 = s(Y_2), Z_2 = 0}
natural_number(s(0))
natural_number(s(0))
plus(X_2, Y_2, 0)
%2 { X_3 = 0 }
%2 { X_3 = 0 }
%3 { X_2 = 0, Y_2 = 0, X_3 = 0 }
natural_number(0)
natural_number(0)
natural_number(0)
%1 {}
%1 {}
%1 {}
true
true
true