מבחן בקורס: עקרונות שפות תכנות, 202-1-2051
מועד: א
תאריך: 23/6/2021
שמות המרצים: מני אדלר, בן אייל, מיכאל אלחדד, ירון גונן
מיועד לתלמידי: מדעי המחשב והנדסת תוכנה, שנה ב', סמסטר ב'
משך המבחן: 3 שעות
חומר עזר: אסור

הנחיות כלליות:
- יש לענות על כל השאלות בגיליון התשובות. מומלץ לא לחרוג מן המקום המוקצה.
- אם אינכם יודעים את התשובה, ניתן לכתוב 'לא יודע' ולקבל 20% מהניקוד על הסעיף/השאלה.

שאלה 1: תחביר וסמנטיקה ____________________ 30 נק
שאלה 2: מערכת טיפוסים ____________________ 25 נק
שאלה 3: מבני בקרה ____________________ 30 נק
שאלה 4: תכנות לוגי ____________________ 20 נק

סה"כ ________________________________ 105 נק

בהצלחה!

שאלה 1: תחביר וסמנטיקה [30 נקודות]

א. השלימו את הכתובות הלקסיקליות של המשתנים בקוד הבא: [6 נק']

```
(define g 1)

(lambda (a b c)
 (if (eq? [b: __  __] [g: __  __])
     ((lambda (c)
        (cons [b: __  __] [c: __  __]))
      [a: __  __])
     [b: __  __]))
```

ב. הקוד הבא (שראינו בכיתה) מדגים במודל ההחלפה כיצד ללא renaming החישוב שגוי. בדוגמה זו יש שימוש במשתנה גלובלי z, להדגמת הבעיה:

```
(define z 1)

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

תנו דוגמת קוד נוספת המדגימה את הבעיה, **שאינה משתמשת במשתנה גלובלי** [6 נק']

ג. הסבירו (ללא קוד) כיצד ניתן להשתמש בכתובות לקסיקליות כך שלא יהיה צורך ב-renaming **במודל ההחלפה**. הבחינו בין משתנים גלובליים ו-לא גלובליים בארגומנט שמוחלף ב-body של ה-closure (רמז: מה הבעיה שבגללה נדרש renaming, ואיך כתובות לקסיקליות פותרות אותה) [10 נק']

ד. הסבירו (ללא קוד) כיצד ניתן להשתמש בכתובות לקסיקליות **במודל הסביבות** על מנת לבצע גישה ישירה למשתנה בסביבה, ולא לחפש אותו בסביבות העוטפות [8 נק']

שאלה 2: טיפוסים [25 נקודות]

2.1 Typing Statements [4 נק]

עבור ה-type statements הבאים, רשמו האם הם נכונים, ואם לא למה.

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

____

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

____

2.2 Type Inference

2.2.1 [18 נקודות] כתבו את רשימת משתני טיפוס ורשימת המשוואות הנגזרות כאשר מריצים את האלגוריתם של הסקת טיפוסים על הביטוים הבאים (אין צורך לפתור את המשוואות).
עבור כל תת-ביטוי ברשימת משתני הטיפוס רשמו את ה-type של הביטוי לפי הגדרת ה-AST.

נתון ה-typing rule עבור ביטוי מסוג `define-exp`:

```
Typing rule define:
  For every: type environment _Tenv,
             variable declaration _x1
             expressions _e1 and
             type expressions _S1:
  If _Tenv o {_x1 : _S1} |- _e1 : _S1
  Then _Tenv |- (define _x1 _e1) : void
```

לדוגמא עבור הביטוי:

```
(L5
  (define g (lambda (f x) (f x)))
  (g + 4))
```

```
Expression                          Variable    AST Type
================================    ========    =========
(L5 …)                              T0          Program
(define g (lambda …))               T1          Define-Exp
(lambda (f x) (f x))                T2          Proc-Exp
(f x)                               T3          App-Exp
f                                   Tf          VarRef
x                                   Tx          VarRef
(g + 4)                             T4          App-Exp
g                                   Tg          VarRef
+                                   T+          PrimOp
4                                   Tnum4       Num-Exp
```

Construct type equations:

```
Expression                          Equation
================================    ============================
(L5 …)                              T0 = T4
(define g (lambda …))               T1 = void
                                    Tg = T2
(lambda (f x) (f x))                T2 = [Tf * Tx -> T3]
(f x)                               Tf = [Tx -> T3]
(g + 4)                             Tg = [T+ * Tnum4 -> T4]
+                                   T+ = [Number -> Number]
4                                   Tnum4 = Number
```

ציינו את הטיפוסים והגדירו את המשוואות עבור התכנית הבאה:

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

```
Expression                              Variable   AST Type
======================================  ========   =========
(L5 …)                                  T0         Program
...
```

```
Expression                      Equation
==============================  ===========================
(L5 …)                          T0 = ...
...
```

2.2.2 [3 נק]

כתבו את ה-type הנגזר עבור g בתוכנית:

שאלה 3: מבני בקרה [30 נקודות]

א. נתונה הפונקציה `add-at-end` אשר משרשרת איבר לסוף רשימה, ומחזירה את הרשימה החדשה:

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

המירו את הפונקציה לגרסת CPS:

[5 נקודות]

```
;; 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 (lambda (x) x)) => '(1 2 3 4)
(define add-at-end$
 (lambda (lst a cont)
```

____

____

____

____

____

____

הוכיחו כי הפונ' `add-at-end` ו-`add-at-end$` שקולות CPS

[10 נקודות]

____

____

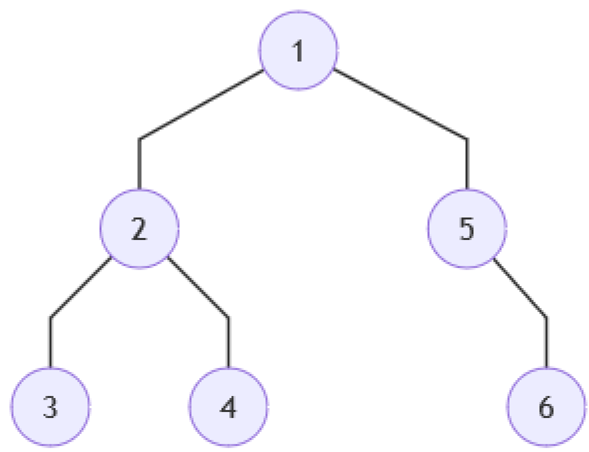
____

____

____

____

ב. כתבו בשפת Scheme את הפונקציה `$tree-reduce` המבצעת reduce על עץ בינארי בסדר pre-order.
הפונקציה מקבלת $reducer (פונקציית CPS בינארית אשר מקבלת צובר acc ואיבר נוכחי cur), ערך התחלתי, עץ בינארי ו-continuation. בשאלה זו, עצים בינאריים ב-Scheme מיוצגים בצורת רשימה בה האיבר הראשון הוא שורש העץ, האיבר השני הוא תת-העץ השמאלי, והאיבר השלישי הוא תת-העץ הימני. לדוגמה, העץ הבינארי:



מיוצג ע"י הרשימה:

```
'(1 (2 (3 () ()) (4 () ())) (5 () (6 () ())))
```

דוגמאות הרצה על העץ הנ"ל:

```
> (define id (lambda (x) x))
> (tree-reduce$ (lambda (acc cur cont)
                  (cont (string-append acc (number->string cur))))
                "" tree id)
"123456"

> (tree-reduce$ (lambda (acc cur cont) (cont (+ acc cur))) 0 tree id)
21
```

השתמשו בפונקציות הממשק הבאות לגישה לעץ:

```
(empty? tree)         ;; True if tree is '()
(tree->data tree)     ;; Returns the value of the node
(tree->left tree)     ;; Returns the left subtree of tree
(tree->right tree)    ;; Returns the right subtree of tree
```

[15 נקודות]

```
;; Signature: tree-reduce$(reducer$, init, tree, cont)
;; Type: ______________________________________________
(define tree-reduce$
  (lambda (reducer$ init tree cont)
```

שאלה 4: תכנות לוגי [20 נקודות]

א. ממשו את הפרדיקט take/3. רשימה List, מספר צ'רץ' N ותת-רשימה Sublist עומדים ביחס אם Sublist היא N האיברים הראשונים של List. אם יש פחות מ-N איברים ב-List, תת-הרשימה Sublist תהיה List כולה.

לדוגמה:

```
?- take([1, 2, 3, 4, 5], s(s(s(0))), X).
X = [1, 2, 3] ;
false.

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

[5 נקודות]

ב. ממשו את הפרדיקט pad/3. רשימה List, מספר צ'רץ' N ורשימה Padded עומדים ביחס אם Padded היא הרשימה List בתוספת כוכביות (הביטוי *) לפי הצורך, כדי להגיע לאורך N. אם N קטן מאורך הרשימה, אין צורך בריפוד.

לדוגמה:

```
?- pad([i, love, ppl], s(s(s(s(s(0))))), X).
X = [i, love, ppl, *, *] ;
false.

?- pad([i, love, ppl], s(0), X).
X = [i, love, ppl] ;
false.

% Signature: pad(List, N, Padded)/3
% Purpose: Padded is List padded with *s to reach length N
```

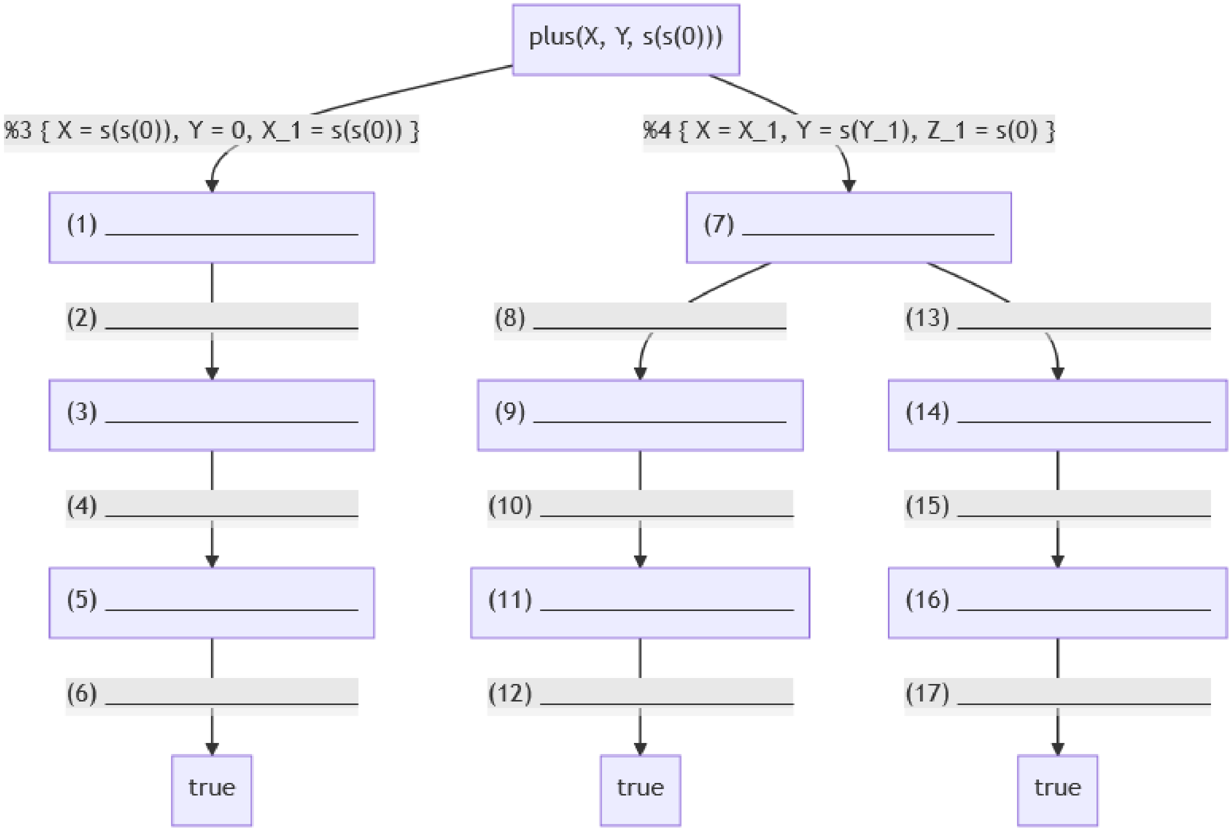
[5 נקודות]

ג. בהינתן הפרוצדורות הבאות:

```
% Signature: natural_number(X)/1
% Purpose: X is a natural number in Church encoding
natural_number(0). %1
natural_number(s(X)) :- natural_number(X). %2

% Signature: plus(X, Y, Z)/3
% Purpose: Z = X + Y
plus(X, 0, X) :- natural_number(X). %3
plus(X, s(Y), s(Z)) :- plus(X, Y, Z). %4
```

השלימו את עץ ההוכחה הבא עבור השאילתה:

```
?- plus(X, Y, s(s(0))).
```



כתבו ליד כל מספר את ה-substitution/כלל המתאימים. שימו לב לכתוב ליד כל substitution את מספר הכלל הרלוונטי; כפי שמופיע בחלקים המלאים בראש העץ. [10 נקודות]

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

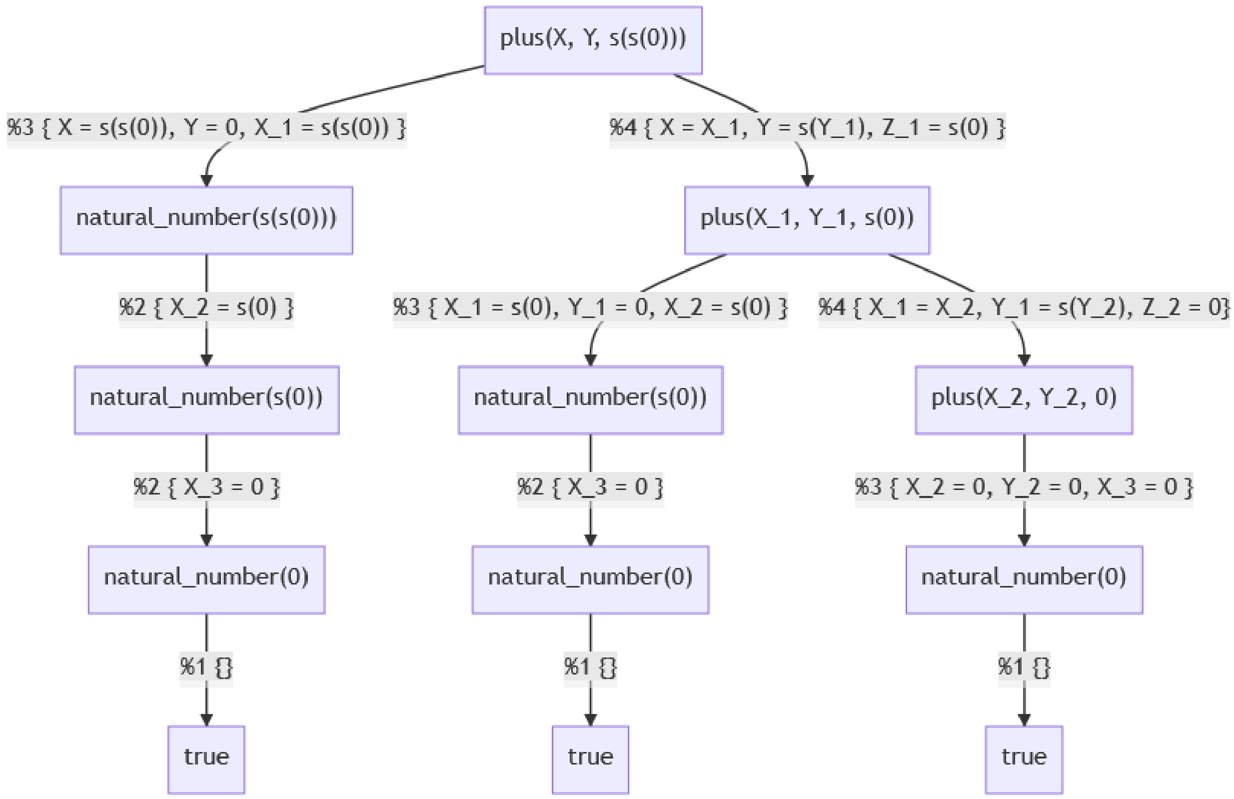
```
% Signature: take(List, N, Sublist)/3
% Purpose: Sublist is the first N elements from List

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