שאלה 1.b

טענה: הפרוצדורה $append שקולה CPS לפרוצדורה append. כלומר, לכל 2 רשימות lst1,lst2 ולכל continuation, אשר יסומן על ידי c מתקיים : ((append$ lst1 lst2 c) = (c (append lst1 lst2

הוכחה: כיוון שהפרוצדורה append היא רקורסיבית, ההוכחה מתבצעת על ידי שימוש באינדוקציה.

נבצע את האינדוקציה על אורך הרשימה lst1 נסמנו n.

בסיס האינדוקציה: n=0 , אז

a-e[(append$ lst1 lst2 c)] →* a-e[(c lst2)] = a-e[(c lst2)]= a-e[(c (append lst1 lst2)]

הנחת האינדוקציה: עבור $n = k \in \mathbb{N}$ הטענה מתקיימת לכל $k \geq i$ כך שi הוא גודל הרשימה 'lst1.

כלומר: ((append$ lst1' lst2 c) = (c (append lst1' lst2)

צעד האינדוקציה: יהא $k \in \mathbb{N}$ ו $n = k + 1$, אזי:

a-e[(append$ lst1 lst2 c)] →* a-e[(append$ (cdr lst1) lst2 (lambda (res) (c(cons (car lst1) res))))] →*

מהנחת האינדוקציה נקבל :

a-e[((lambda (res) (c(cons (car lst1) res))) (append (cdr lst1) lst2))] →*

a-e[(c (cons (car lst1) (append (cdr lst1) lst2))] = a-e[(c (append lst1 lst2))]

מ.ש.ל

שאלה *2.d*

<u>reduce1-lzl</u> – זהו *reduce* הרגיל שמכירים, כלומר בהינתן רשימה סופית, ו*acc* מחזיר את פעולת ה*reducer* עבור כל האיברים ברשימה(בעזרת *lzl* כמובן, שכן מחשבת לפי הצורך). במידה וזו רשימה אינסופית, לעולם לא נעצור ולא נחזיר ערך סופי.

<u>reduce2-lzl</u> - ניתן להשתמש בסוג זה עבור רשימות אינסופיות אשר נרצה לעצור לאחר *n* איברים עבור *n* סופי. פעולה זו עושה את מה ש*reduce1* עושה רק שהיא עובדת גם על רשימות אינסופיות עם חסם עליון.

<u>reduce3-lzl</u> – כאשר נרצה לבצע פעולת *reduce* רגילה על רשימה, אך נרצה גם לראות את החישובים שנוצרים בדרך בין כל מעבר של 2 איברים ואת הערך של ה-*acc*.

שאלה *2.g*

יתרון - בgenerate-pi-approximations אותה מימשנו בעבודה זו אין בעיה של זיכרון (בגלל מימוש *reduce3-lzl* עם רשימה עצלה) ולכן נוכל להגיע לקרובים מדויקים יותר (יותר ספרות) לעומת *pi-sum* שהוא ממומש כרקורסית ראש – אשר פותחת פריים נוסף לחישוב בכל שלב ולכן הזיכרון אוזל.
כמו כן, כאשר יש כמות לא מוגבלת של נתונים וכאשר יש נתונים לא זמינים או שהגישה אליהם יקרה/סבוכה/לא בטוחה, רצוי להשתמש ברשימה עצלה.

חיסרון - החיסרון בgenerate-pi-approximations הוא שהוא ממומש בעזרת טיפוס נתונים חדש (רשימה עצלה) ולכן דורש הכפלה של כל הפעולות. כמו כן מאחר ויש בו שימוש ברשימה עצלה הוא יכול להוות מקור נוסף לשגיאות טיפוס.

שאלה 3.1

```
a. unify[t(s(s), G, s, p, t(K), s), t(s(G), G, s, p, t(K), U)]
```

unify[t(s(s), G, s, p, t(K), s), t(s(G), G, s, p, t(K), U)]
S={}
A*S=t(s(s), G, s, p, t(K), s)
B*S=t(s(G), G, s, p, t(K), U)
S=S*{G=s}={G=s}
A*S=t(s(s), s, s, p, t(K), s)
B*S=t(s(s), s, s, p, t(K), U)
S=S*{U=s}={G=s,U=s}
A*S=t(s(s), s, s, p, t(K), s)
B*S=t(s(s), s, s, p, t(K), s)

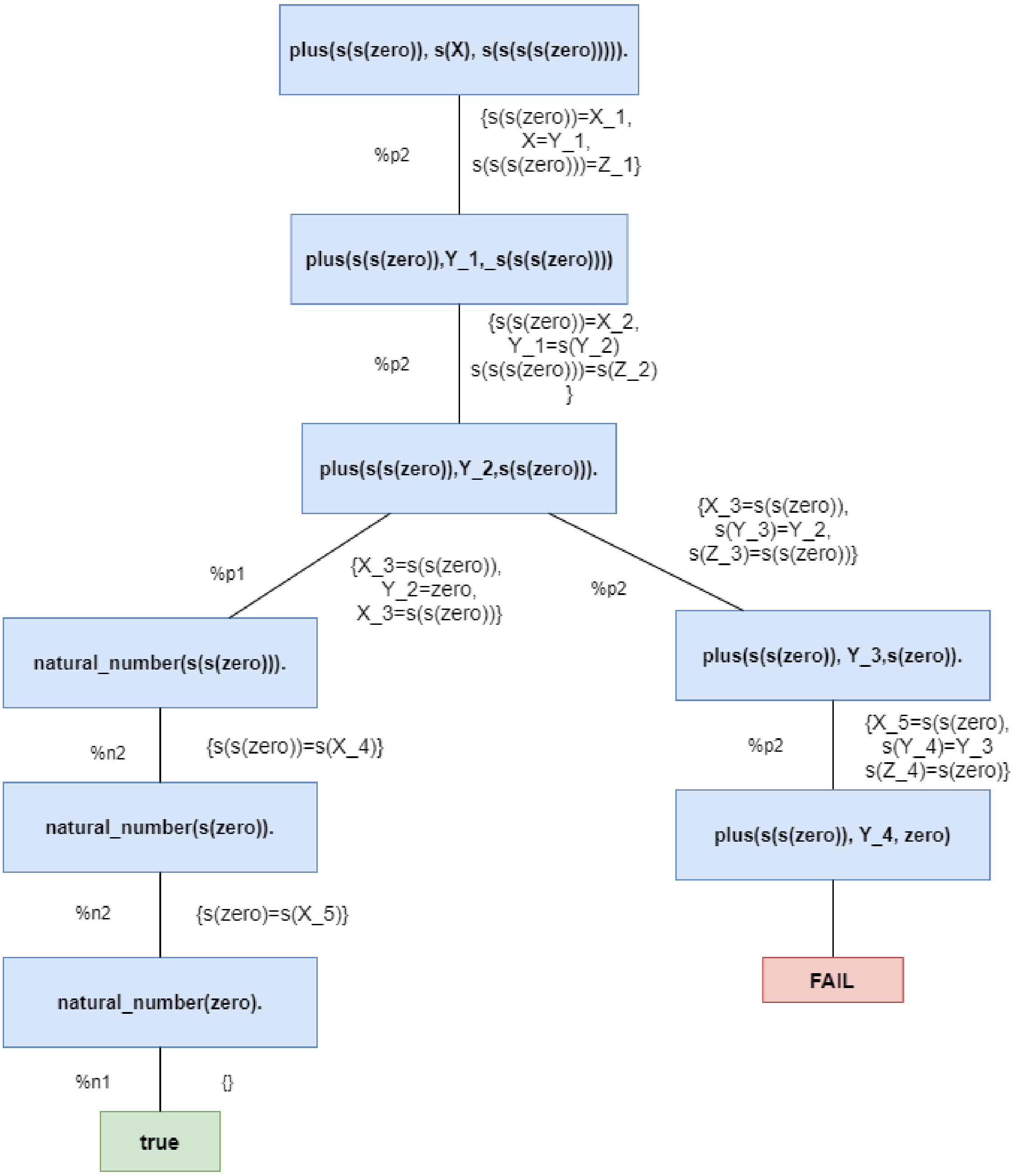
תקין כלומר {G=s,U=s} *הינו ה* unifier *הכללי ביותר* (MGU).

```
b. unify[p([v | [V | W]]), p([[v | V] | W])]
```

unify[p([v | [V | W]]), p([[v | V] | W])]
S={}
A*S=p([v | [V | W]])
B*S=p([[v | V] | W])

לא תקין. נשים לב שהמבנה לא חוקי שכן $v \neq [v|V]$.

שאלה 3.3


plus(s(s(zero)), s(X), s(s(s(s(zero))))).
%p2
{s(s(zero))=X_1,
X=Y_1,
s(s(s(zero)))=Z_1}
plus(s(s(zero)),Y_1,_s(s(s(zero))))
%p2
{s(s(zero))=X_2,
Y_1=s(Y_2)
s(s(s(zero)))=s(Z_2)
}
plus(s(s(zero)),Y_2,s(s(zero))).
%p1
{X_3=s(s(zero)),
Y_2=zero,
X_3=s(s(zero))}
%p2
{X_3=s(s(zero)),
s(Y_3)=Y_2,
s(Z_3)=s(s(zero))}
natural_number(s(s(zero))).
%n2
{s(s(zero))=s(X_4)}
natural_number(s(zero)).
%n2
{s(zero)=s(X_5)}
natural_number(zero).
%n1
{}
true
plus(s(s(zero)), Y_3,s(zero)).
%p2
{X_5=s(s(zero),
s(Y_4)=Y_3
s(Z_4)=s(zero)}
plus(s(s(zero)), Y_4, zero)
FAIL