PPL Assignment 5

Question 1: CPS

1.1

b)

נוכיח באינדוקציה :

הפרוצדורה append$ שקולה CPS ל- append, כלומר לכל $n \in N$ ולכל $cont$ מתקיים

$$append\$\ (lst1\ lst2\ cont) = cont(\ append\ (lst1\ lst2))$$

בסיס האינדוקציה : $n = 0$, משמע אורך הרשימה $|lst1| = 0$

עבור טענה זו נקבל כי $a - e[append\$\ ([\ ]\ lst2\ cont)] ==>^* a - e[(cont\ lst2)] =$

$$a - e[(cont(append\ [\ ]\ lst2))]$$

הנחת האינדוקציה :

עבור $n = k \in \mathbb{N}$ הטענה מתקיימת לכל רשימה באורך $k \geq i$, כלומר ,

$$append\$\ (lst1\ lst2\ cont) = \big(cont\ (append\ lst1\ lst2)\big)\ .$$

צעד האינדוקציה :

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

$$a - e[(append\$\ lst1\ lst\ cont)] ==>^*$$

$$a - e\begin{bmatrix}(append\$\ \ (cdr\ lst1)\ lst2\\ (lambda\ (cont - lst1)\big(cont\ (cons\ (car\ lst1)\ cont - lst1)\big)\big)\end{bmatrix}$$

מהנחת האינדוקציה נקבל :

$$a - e\begin{bmatrix}(lambda\ (cont - lst1)\\ \big(cont\ (cons\ (car\ lst1)\ cont - lst1)\big)\big)(append(cdr\ lst1)(lst2)\end{bmatrix}$$

ומכיוון שהפעולה $cont$ משרשרת את האיבר הראשון לשאר הרשימה , נקבל כי :

$$a - e[cont\ (append\ (lst1)(lst2))]$$

לכן הוכחנו כי לכל $n \in \mathbb{N}$ נקבל $append\$\ (lst1\ lst2\ cont) = \big(cont\ (append\ lst1\ lst2)\big)$ כנדרש.

Question 2: Lazy lists

d) **reduce1-lzl** - In a case I want to get the sum(or any other binary func) of all values. example: I want to count the value of all the coins in my pocket.

reduce2-lzl - In a case I have a long list of numbers and I want to get the sum(or any other binary func) of only part of the values. example: I have a monthly revenue list ordered by date, and I want to count the value of revenue for the first 10 days.

reduce3-lzl - In a case I have a long list of numbers and I want to know for each row of the list, the sum(or any other binary func) of the values till that row. example: I have a monthly revenue list ordered by date, and I want to be in control and to know for every row the revenue till that row.

g) Generate-pi-approximation is using lazy-list evaluation , which means that no memory has to be stored somewhere , and even an infinity list can represent that way.
Furthemore, Lazy evaluation such Generate-pi-approximation,
is often combined with memoization, After a function's value is computed for that parameter or set of parameters, the result is stored in a lookup table that is indexed by the values of those parameters.
The next time the function is called, the table is consulted to determine whether the result for that combination of parameter values is already available.
If so, the stored result is simply returned. If not, the function is evaluated and another entry is added to the lookup table for reuse.
That is why this Implementation is better than the Pi-sum one.
Although those facts, In the Pi-sum Implementation no Laziness methodology is used,
but if you want to get only the values from the k'th to n'th indexes , you could do it,
while in the Generate-pi-approximation you aren't able to do so.

Question 3: Logic programing

3.1

a) 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)

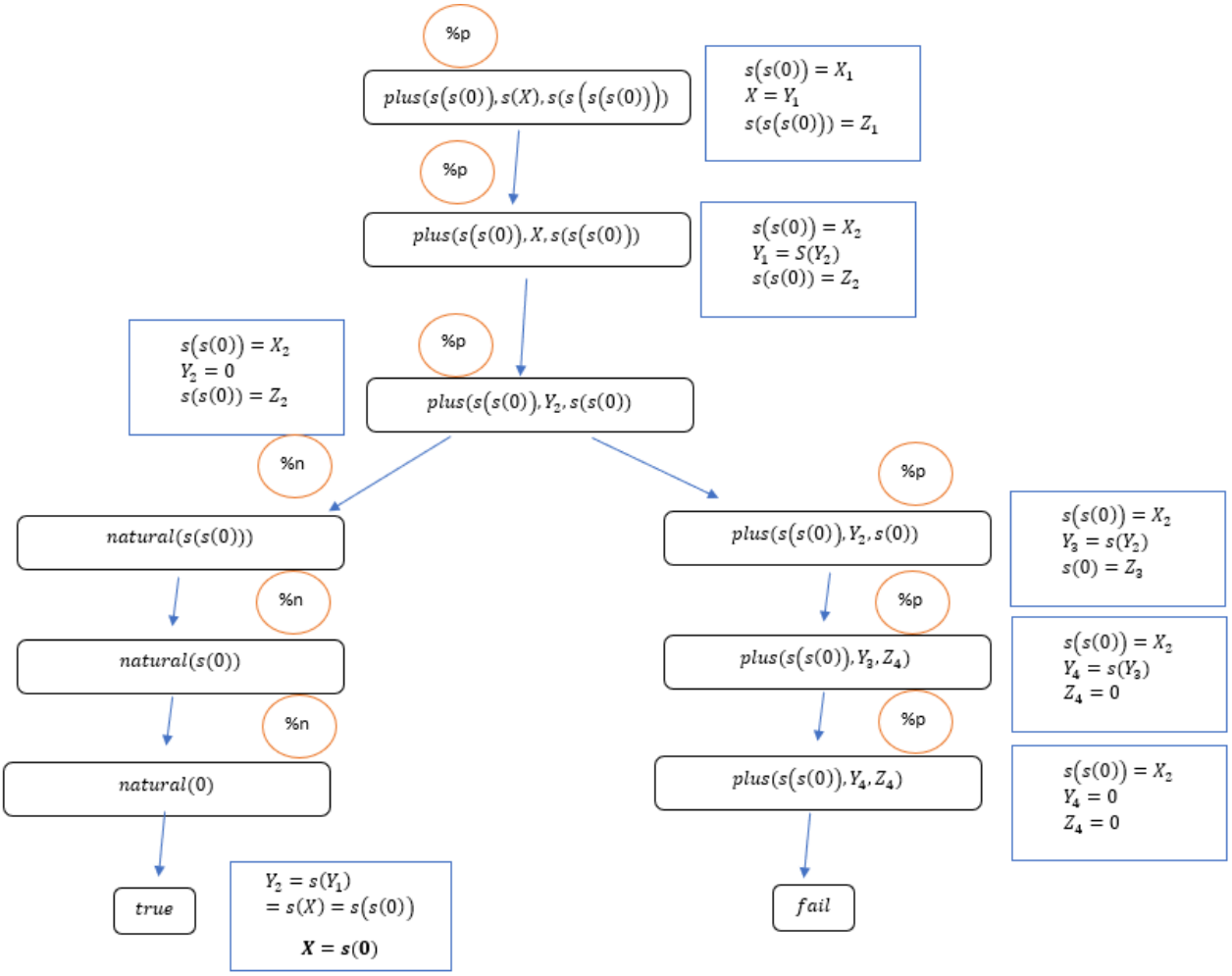
⇒ **S = {G = s, U = s}**

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

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

⇒ **FAIL: v != [v | V] not the same structure**

3.3


%p
plus(s(s(0)), s(X), s(s(s(s(0)))))
s(s(0)) = X1
X = Y1
s(s(s(0))) = Z1
%p
plus(s(s(0)), X, s(s(s(0))))
s(s(0)) = X2
Y1 = S(Y2)
s(s(0)) = Z2
%p
plus(s(s(0)), Y2, s(s(0)))
s(s(0)) = X2
Y2 = 0
s(s(0)) = Z2
%n
natural(s(s(0)))
%n
natural(s(0))
%n
natural(0)
true
Y2 = s(Y1)
= s(X) = s(s(0))
X = s(0)
%p
plus(s(s(0)), Y2, s(0))
s(s(0)) = X2
Y3 = s(Y2)
s(0) = Z3
%p
plus(s(s(0)), Y3, Z4)
s(s(0)) = X2
Y4 = s(Y3)
Z4 = 0
%p
plus(s(s(0)), Y4, Z4)
s(s(0)) = X2
Y4 = 0
Z4 = 0
fail