PPL – assignment 5

Part 1:

Question 1.1 a

(The code is also in ex5.rkt it was added here because it is relevant to part b)

Question 1.1 b

We will prove append$ is CPS-equivalent to append by induction on list1’s length   
(indicted by *n*). To do so we will prove that:

For every list1, list2 lists, and any continuation function (named *cont* for short)   
append$ is CPS-equivalent to append.

i.e.:

**Induction basis:** n=0, list1 is empty.

**Induction assumption:** let us assume, that for , s.t list1 length is n,   
the equation is valid:

**Induction step:** we will prove that the equation is valid for n+1:

From the assumption of induction for (with length n):

Part2:

Question 2 d

**The reduce1-lzl:** is used for basic reduce functionality using finite lazy-list, giving the same utility as regular reduce function used for normal lists.

**The reduce2-lzl:** is most useful for computations which involve infinite lazy-list, in which we know exactly how many items we want to compute. Also, can be used to get the reduce functionality over sub lazy list of known length.

**The reduce3-lzl:** is useful for cases in which the intermediate accumulated calculations are necessary, a good example of such case is for approximations calculation   
e.g., generate-pi-approximations in part f.

Question 2 g

**Advantage of implementation via *generate-pi-approximations* over *pi-sum*:**

- If we look at the implementation of pi-sum taught in class we can see that it's recursion not in tail form, on the over hand *generate-pi-approximations*can be implemented as a tail form recursion. Using tail call optimization pi-sum can cause stack overflow as new frames continuously open while *generate-pi-approximations* will be performed as iteration and not face the same fate.

- If we wanted to duplicate *generate-pi-approximations* result for n items using *pi-sum* we will need to perform it n times. In cases such as this when intermediate results are required *generate-pi-approximations* is significantly more efficient than *pi-sum*.

- Because *generate-pi-approximations* returns the results as lazy list it benefits from the same advantages of lazy list such as the ability to delay computation of next value and the ability to perform high order functions for lazy lists (e.g., map-lzl) on the results, which can be used to perform efficient computation one value at a time.

**Advantage of implementation via *pi-sum* over *generate-pi-approximations*:**

- generate-pi-approximations start from the begging at each stage i.e. the first approximations computed using one term, the second computed using the two terms as so on, unlike pi-sum which compute the approximations in a straightforward way.

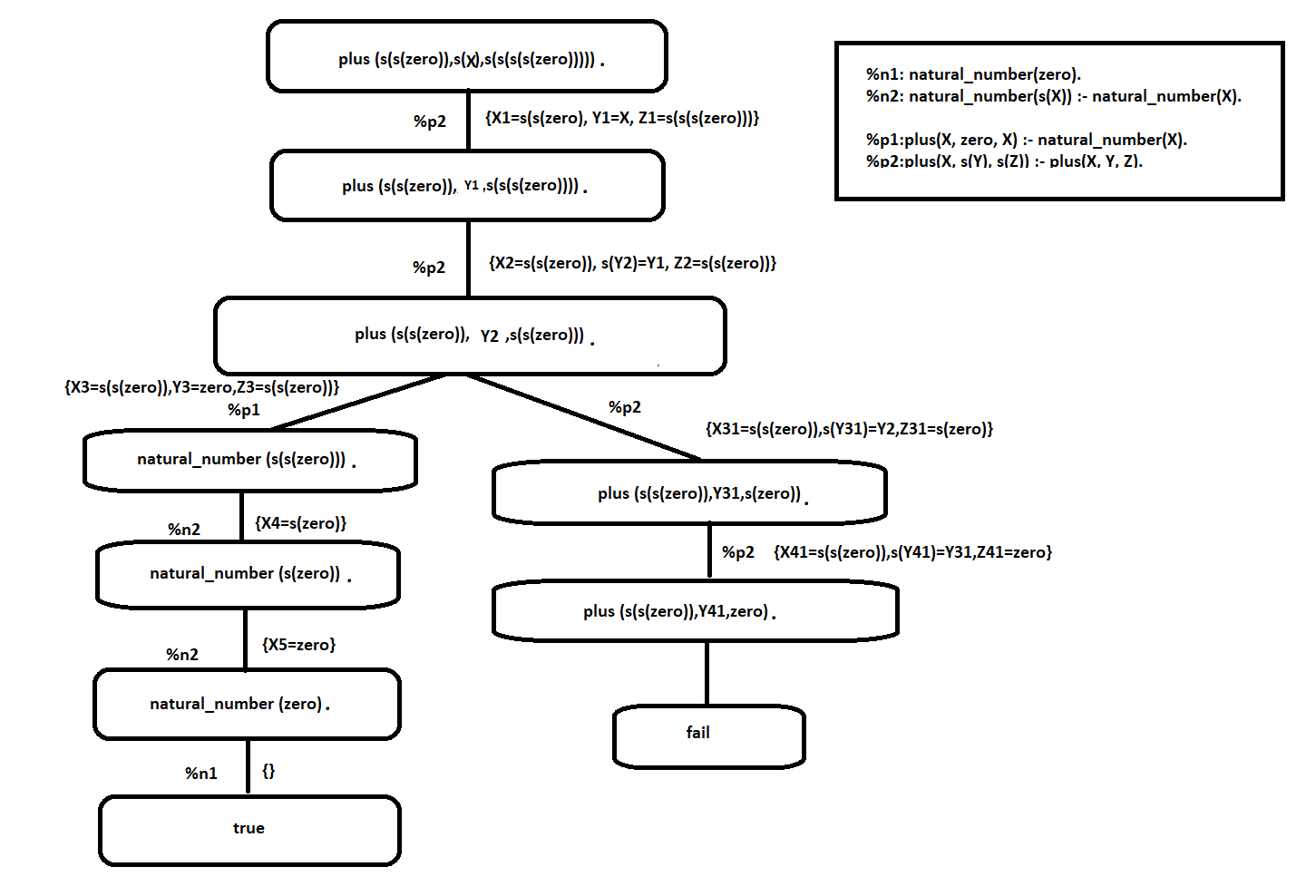
- generate-pi-approximations is implemented using several high order functions which can cause overhead.

Part 3:

Question 1 a (

Question 1 b

Question 3

****