Question 1.1.b

We must prove that (append$ lst1 lst2 cont) = (cont (append lst1 lst2)) by induction on the length of lst1

Let the length of lst1 be n.

Base: n = 0

applicative-eval[(append$ '() lst2 cont)]

→ applicative-eval[(cont lst2)]

From the definition of append:

→ applicative-eval[(cont (append '() lst2))]

Induction assumption: assume that for any lst1 of length n-1 the claim holds,

i.e., applicative-eval[(append$ lst1 lst2 cont)] = applicative-eval[(cont (append lst1 lst2))]

Induction step: let lst1 be a list of length n, then

applicative-eval[(append$ lst1 lst2 cont)]

→ applicative-eval[(append$ (cdr lst1) lst2 (lambda (res) (cont (cons (car lst1) res))))]

From the Induction assumption:

→ applicative-eval[((lambda (res) (cont (cons (car lst1) res)))) (append (cdr lst1) lst2)]

→ applicative-eval[(cont (cons (car lst1) (append (cdr lst1) lst2)))]

From the definition of append:

→ applicative-eval[(cont (append lst1 lst2))]

Question 2.d

In cases where we deal with finite lazy lists, we can use reduce1-lzl with out the fear of being stuck in infinite loop or infinite recursion.

In cases where we are dealing with infinite lazy lists, and we know the exact number of elements we want to reduce in advance we can use reduce2-lzl.

In cases where we are dealing with infinite lazy lists, and we do not know the exact number of elements that we want to reduce (for example if we want to continue reducing until reaching a goal), we can use reduce3-lzl.

Examples for the uses:

reduce1-lzl → summing a list of 10 integers.

reduce2-lzl → summing the first 10 integers in a list of all integers.

reduce3-lzl → summing integers until reaching a number greater than 321561313.

Question 2.g

First of all, since we return the series (sums) as a lazy list, thus we can always reach better approximation by getting the next element in the lazy list, which says (and as we implemented it) we can return kind of infinite series and thus getting better approximation (this may help when we do not have certain accuracy that we want to reach).

In addition, we can reach approximations that the pi-sum function can not reach since the pi-sum was written as a recursive function, and thus in some point it will reach to a full calling stack (which is not true in the case of the lazy list, since we do not have to keep old "links" in memory).

The disadvantage is that in order to reach the n'th element in the lazy list we must go over all the n-1 elements that comes before it (can be solved by writing a procedure to do the work).

Question 3.1.a

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

step 0: sub = {}

equations = [t(s(s), G, s, p, t(K), s) = t(s(G), G, s, p, t(K), U)]

step 1: eq: t(s(s), G, s, p, t(K), s) = t(s(G), G, s, p, t(K), U)
eq o sub: t(s(s), G, s, p, t(K), s) = t(s(G), G, s, p, t(K), U)
we have two predicates with the same symbols and number of arguments, then we add the following equations to the equations: [s(s) = s(G), G = G, s = s, p = p, t(K) = t(K), s = U]
→ sub = {}
equations = [s(s) = s(G), G = G, s = s, p = p, t(K) = t(K), s = U]

step 2: eq: s(s) = s(G)
eq o sub: s(s) = s(G)
we have two predicates with the same symbols and number of arguments, then we add the following equations to the equations: [s = G]
→ sub = {}
equations = [G = G, s = s, p = p, t(K) = t(K), s = U, s = G]

step 3: eq: G = G
eq o sub: G = G
one side is a variable, and the other side is the same variable, continue.
→ sub = {}
equations = [s = s, p = p, t(K) = t(K), s = U, s = G]

step 4: eq: s = s
eq o sub: s = s
both sides are atomic, and they are the same, continue.
→ sub = {}
equations = [p = p, t(K) = t(K), s = U, s = G]

step 5: eq: p = p
eq o sub: p = p
both sides are atomic, and they are the same, continue.
→ sub = {}
equations = [t(K) = t(K), s = U, s = G]

step 6: eq: t(K) = t(K)
eq o sub: t(K) = t(K)
we have two predicates with the same symbols and number of arguments, then we add the following equations to the equations: [K = K]
→ sub = {}
equations = [s = U, s = G, K = K]

step 7: eq: s = U
eq o sub: s = U
one side is a variable, and the other side is a term.
→ sub = { U = s }
equations = [s = G, K = K]

step 8: eq: s = G
eq o sub: s = G
one side is a variable, and the other side is a term.
→ sub = { U = s, G = s}
equations = [K = K]

step 9: eq: K = K
eq o sub: K = K
one side is a variable, and the other side is the same variable, continue.
→ sub = { U = s, G = s}
equations = []

The equations list is empty; we stop here and return the mgu { U = s, G = s }

Question 3.1.b

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

step 0: sub = {}

equations = [p([v | [V | W]]) = p([[v | V] | W])]

step 1: eq: p([v | [V | W]]) = p([[v | V] | W])
eq o sub: p([v | [V | W]]) = p([[v | V] | W])
we have two predicates with the same symbols and number of arguments, then we add the following equations to the equations: [[v | [V | W]] = [[v | V] | W]]
→ sub = {}
equations = [[v | [V | W]] = [[v | V] | W]]
to make the question clearer we decided to convert the lists to "cons" terms, thus the equations will be: [cons(v, cons(V, cons(W, []))) = cons(cons(v, cons(V, [])),cons(W, []))]

step 2: eq: cons(v, cons(V, cons(W, []))) = cons(cons(v, cons(V, [])),cons(W, []))
eq o sub: cons(v, cons(V, cons(W, []))) = cons(cons(v, cons(V, [])),cons(W, []))
we have two predicates with the same symbols and number of arguments, then we add the following equations: [v = cons(v, cons(V, [])), cons(V, cons(W, [])) = cons(W, [])]
→ sub = {}
equations = [v = cons(v, cons(V, [])), cons(V, cons(W, [])) = cons(W, [])]

step 3: eq: v = cons(v, cons(V, []))
eq o sub: v = cons(v, cons(V, []))

we got an equation where one side is atomic and the other side is a predicate, thus none of the three cases hold, thus we return FAIL.

Question 3.3

Near each arrow is the appropriate substitution.

At the end by composing all the substitutions we get that X=s(zero).