Assignment 5

Q1

Q1.1.a

First of all, we'll go over 'take' function we saw in class.

```
;; Signature: take(lz-lst,n)

;; Type: [LzL*Number -> List]

;; If n > length(lz-lst) then the result is lz-lst as a List

(define take

  (lambda (lz-lst n)

    (if (or (= n 0) (empty-lzl? lz-lst))

      empty-lzl

      (cons (head lz-lst)

            (take (tail lz-lst) (- n 1))))))
```

lazy list lzl1 and lazy list lzl2 are equivalent if for every $n \geq 0$ (n is a natural number) (take lzl1 n) = (take lzl2 n).

Q1.1.b

We will prove the equivalence of even-squares-1 and even-squares-2 by induction.

In other words, we will prove that (take even-squares-1 n) = (take even-squares-2 n)

base case: $n = 0$. (take even-squares-1 0) returns '(), and (take even-squares-1 0) return '() as well.

Induction hypothesis: We assume that the statement holds for k where n > k > 0.

Induction step: We will prove that the statement holds for n=k.

We'll prove that (take even-squares-1 n) = (take even-squares-2 n).

Auxiliary claim: for x, a natural number, $isEven(x^2) = isEven(x)$, where isEven is predicate that checks if some natural number n is even (i.e. checks if n % 2 == 0).

We'll separate to 2 cases:

case 1: x is even. So, isEven(x) will return true, and x^2 is x * x and we know that multiplication of even numbers is even result. So, for $isEven(x^2)$ we'll return true also.

case 2: x is odd. So, isEven(x) will return false, and x^2 is x * x and we know that multiplication of odd numbers is odd result. So, for $isEven(x^2)$ we'll return false either.

We prove the equivalence of $isEven(x^2) = isEven(x)$.

Back to our induction.

When we apply (take even-squares-1 n), as defined in the function take, we're doing cons between (head even-sqaures-1) and (take (tail even-sqaures-1) (n-1)). By the induction hypothesis, (take (tail even-sqaures-1) (n-1)) = (take (tail even-sqaures-2) (n-1)). So, we need to prove that (head even-sqaures-1) =

(head even-sqaures-2). We'll check if the head of both lists is equal. Suppose the first element before filtering and squaring called x. So, in even-sqaures-1 x first squared and then filtered by isEven, so we're squaring x and we get x^2 and then filtering it by isEven, so x^2 (and x himself) has to be even number (if not it won't be the head of the list).

On the other hand, we have even-sqaures-2, which first filtered x by isEven, so x has to be even number, and then we squared it, so we get x^2, where x is an even number.

By our auxiliary claim we can conclude that head of both lists is equal. Therefore, both lazy lists are equal, by my definition of equivalence between two lazy lists.

Q2

Q2.a

A procedure and its Success-Fail-Continuations version are equivalent if for every input value $x_1, x_2, \dots, x_n$ and every Success-Fail-Continuations (call it success and fail):

(f$ $x_1, x_2, \dots, x_n$ success fail) = (success (f $x_1, x_2, \dots, x_n$))

and if (f $x_1, x_2, \dots, x_n$) fails, so fail-continuation procedure is applying.

Q2.d

We'll prove that get-value and get-value$ are equivalent according to the definition.

We'll prove by induction on number of elements in assoc-list, n.

(get-value$ assoc-list key success fail) = (success (get-value assoc-list key))

and if (get-value assoc-list key) fails, so fail-continuation procedure is applying.

base case: $n = 0$. So assoc-list is '(). So (get-value assoc-list key) fails and returns 'fail because assoc-list is empty, and get-value$ applies fail-continuation procedure, as expected.

Induction hypothesis: We assume that the statement holds for k where n > k > 0.

Induction step: We will prove that the statement holds for n = k.

We'll separate to cases:

case 1: key **is not in** assoc-list.

In get-value, the test expression of checking if the key is equal won't satisfy, so we call *(get-value (cdr assoc-list) key)*, until assoc-list will be empty and get-value return 'fail (when checking if accos-list is empty).

We'll observe that in get-value$ the situation will be the same. The test expression of checking if the key is equal won't satisfy, so we call *(get-value$ (cdr assoc-list) key (lambda (get-value-cdr) (success get-value-cdr)) fail)*, until assoc-list will be empty, and fail-continuation procedure will be applied.

So, we can see that in this case, the definition of equivalence is valid.

case 2: key **in** assoc-list.

At first, get-value procedure checks if test expression is satisfied (i.e. the key of the pair in the head of assoc-list is equal to key)

We'll separate to cases:

> **case a:** key of the pair in the head of assoc-list **is equal** to key input.
>
> In this case, the test expression is satisfied and we return the value of that key (call it x). Similarly, get-value$ returns application expression of success on x, i.e. (success x).
>
> So, by the definition we get (success x) = (get-value$ assoc-list key success fail) and also true that (success (get-value assoc-list key)) = (success x).
>
> **case b:** key of the pair in the head of assoc-list **is not equal** to key input.
>
> So, the test expression is not satisfied. Therefore, in get-value we call *(get-value (cdr assoc-list) key)* and in get-value$ we call *(get-value$ (cdr assoc-list) key (lambda (get-value-cdr) (success get-value-cdr)) fail)*. By induction hypothesis, those expressions are equivalent according to the definition.

We prove that in all cases the equivalence is satisfied.

Q3

Q3.1

a. Unify [t (s (s) , G , H , p, t (E) , s) ,

t (s (H) , G , p , p, t (E) , K)]

Failure. According to the algorithm, we see that the functors are equal, so we split to equations of the arguments, and we get:
s (H) = s (s)
H = p
K = s
On the first equation (s (H) = s (s)) we can see that the functors is equal on both sides, so we split to equations of the arguments, and we get:
H = p
K = s
H = s
We add the equation to the substation-set { H = p }.
We apply substitution-set on the next equation – we do nothing.
We add the equation to the substation-set { H = p, K = s}.
We apply substitution-set on the next equation – we get p = s. Both sides are atomic, and not equal, therefore we return **FAIL**.

b. Unify [g (c , v (U) , g , G , U , E , v (M)) ,
g (c , M , g ,v (M) , v (G) , g , v (M))]

Failure. According to the algorithm, we see that the functors are equal, so we split to equations of the arguments, and we get:
M = v (U)
G = v (M)
U = v (G)
E = g

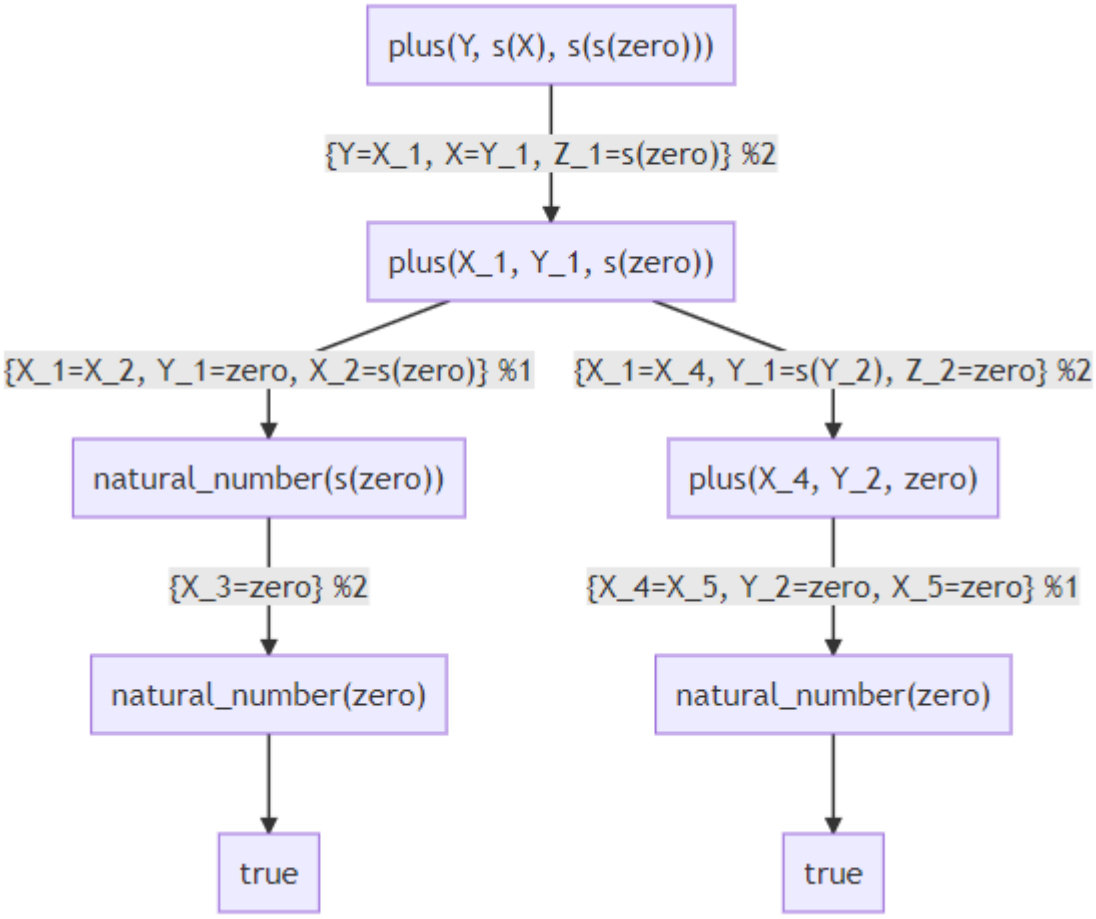
Now our substitution-set is: { M = v (U) }.
We apply the substitution-set on the equation G = v (M), and we get G = v (v (U)).
We add this binding to the substitution-set {M = v (U), G = v (v (U)) }.
We apply the substitution-set on the equation U = v (G), and we get U = v (v (v (U))).
We do **occurs check** and we **FAIL** because U is occur on both sides of the equation.

c. Unify [s ([v | [[v | V] | A]]) ,
s ([v | [v | A]])]

Failure. First we convert to:

Unify [s(cons(v, cons(cons(v, V), A))),
s(cons(v, cons(v, A)))]
According to the algorithm, we can see that functors are equal (s=s), so we split to equations of the arguments, so we get:
cons(v, cons(cons(v, V), A)) = cons(v, cons(v, A))]
We do the same thing on the functor const, and we get:
cons(cons(v, V), A) = cons(v, A).
We do another iteration of the algorithm and we get:
cons(v, V) = v
Now we can notice that there is first disagreement, and because the functor v is not equal to functor cons, we return **FAIL**, according to the algorithm.

Q3.3.a



{Y=X_1, X=Y_1} ○ {X_1=X_2, Y_1=zero, X_2=s(zero)} ○ {X_3=zero} =

{Y=s(zero), X= zero}

{Y=X_1, X=Y_1} ○ {X_1=X_4, Y_1=s(Y_2), Z_2=zero} ○ {X_4=X_5, Y_2=zero, X_5=zero} =

{Y= zero, X= s(zero)}

Q3.3.b

{Y=s(zero), X= zero}

{Y= zero, X= s(zero)}

Q3.3.c

According to the definition, success proof tree is a proof tree that has at least one success path. We can see that the proof tree we draw in 3.3.a has 2 success paths, therefore **this is success proof tree.**

Q3.3.d

We can see that the proof tree we draw in 3.3.a is **finite tree**, because there are finite number of nodes.